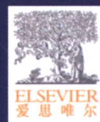


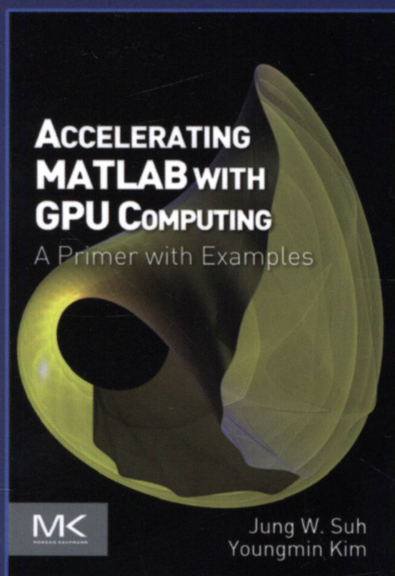
国际信息工程先进技术译丛



GPU与MATLAB混合编程

Accelerating MATLAB with GPU Computing: A Primer with Examples

[韩] 郑郁旭（Jung W. Suh）
金英民（Youngmin Kim） 著
熊磊 李丞 译



- 本书由数据密集计算领域的资深专家撰写
- 以应用实例为主线，提供大量的源代码
- 教你轻松掌握GPU与MATLAB混合编程





机械工业出版社
CHINA MACHINE PRESS

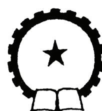
VISIT…

LANZAROTE Caliente.com

国际信息工程先进技术译丛

GPU与MATLAB混合编程

［韩］郑郁旭（Jung W·Suh）
金英民（Young min kim）编著
熊　磊　李　丞　译



机 械 工 业 出 版 社

本书介绍 CPU 和 MATLAB 的联合编程方法，包括不使用 GPU 实现 MATLAB 加速的方法；MATLAB 和计算统一设备架构（CUDA）配置通过分析进行最优规划，以及利用 c-mex 进行 CUDA 编程的方法；MATLAB 与并行计算工具箱和运用 CUDA 加速函数库的方法；计算机图形实例和 CUDA 转换实例。本书通过大量的实例、图示和代码，深入浅出地引导读者进入 GPU 的殿堂。通过阅读本书，读者可以轻松学习使用 GPU 进行并行处理，实现 MATLAB 代码的加速，提高工作效率，从而将更多的时间和精力用于创造性工作和其他事情。

本书可作为相关专业高年级本科生和研究生的教材，也可作为工程技术人员的参考书。

<*Accelerating MATLAB with GPU Computing: A Primer with Examples*>,
< *Jung W.Suh ,Youngmin Kim* >
ISBN: 978-0-12-408080-5(ISBN of original edition)
Copyright © *2014* by Elsevier. All rights reserved.
Authorized Simplified Chinese translation edition published by the Proprietor.
Copyright © *2016* by Elsevier (Singapore) Pte Ltd and China Machine Press.
All rights reserved.
Published in China by < China Machine Press > under special arrangement with Elsevier (Singapore) Pte Ltd.. This edition is authorized for sale in China only, excluding Hong Kong SAR and Taiwan. Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.
本书简体中文版由Elsevier (Singapore) Pte Ltd.授予机械工业出版社在中国大陆地区（不包括香港、澳门特别行政区以及台湾地区）出版与发行。未经许可之出口，视为违反著作权法，将受法律之制裁。
本书封底贴有Elsevier防伪标签，无标签者不得销售。
北京市版权局著作权登记 图字：01-2015-3498

图书在版编目（CIP）数据

GPU 与 MATLAB 混合编程/（韩）郑郁旭，（韩）金英民编著；熊磊，李丞译. — 北京：机械工业出版社，2016.1
（国际信息工程先进技术译丛）
书名原文：Accelerating MATLAB with GPU Computing：A Primer with Examples
ISBN 978-7-111-52904-0
Ⅰ. ①G… Ⅱ. ①郑… ②金… ③熊… ④李… Ⅲ. ①图象处理-程序设计 Ⅳ. ①TP391.41
中国版本图书馆 CIP 数据核字（2016）第 024726 号
机械工业出版社（北京市百万庄大街 22 号 邮政编码 100037）
责任编辑：李馨馨 责任校对：张艳霞
责任印制：常天培
北京机工印刷厂印刷（三河市南阳庄国丰装订厂装订）
2016 年 4 月第 1 版 • 第 1 次印刷
169mm×239mm • 13.5 印张 • 262 千字
0001－3000册
标准书号：ISBN 978-7-111-52904-0
定价：59.00元
凡购本书，如有缺页、倒页、脱页，由本社发行部调换
电话服务
服务咨询热线：（010）88361066
读者购书热线：（010）68326294
（010）88379203
网络服务
机 工 官 网：www.cmpbook.com
机 工 官 博：weibo.com/cmp1952
教育服务网：www.cmpedu.com
金 书 网：www.golden-book.com
封面无防伪标均为盗版

译 者 序

当前，在视频处理、电磁场分析、移动通信、生物信息、人工智能、医疗诊断、流体力学等诸多领域，密集计算的需求越来越旺盛。台式计算机 CPU 发展远远无法满足密集计算的要求。而采用大型工作站进行密集计算，不仅成本高昂，应用也十分不便。利用图形处理器（GPU）中众多的流处理器，实现并行计算，可以极大加速程序运行，是当前密集计算的重要方法。

但要熟练使用 GPU 实现程序加速，需要用户具有较好的编程能力，往往令人望而却步。而 MATLAB 功能强大，简单易用，应用非常广泛。本书聚焦于 GPU 和 MATLAB 的混合编程，采用实例教学的方式，并附上大量源代码，全书浅显易懂，以最小的学习成本，让读者掌握 GPU 加速 MATLAB 的方法，非常适合作为 GPU 入门读物。

本书第 1 章介绍了不使用 GPU 而直接实现 MATLAB 程序加速，读者得以初窥程序加速的基本方法。第 2 章介绍了使用 GPU 前需要的 MATLAB 和 CUDA 配置方法，并以二维卷积为例，详细介绍了 GPU 实现程序加速的流程。第 3 章介绍了多种时间分析工具，引领读者通过时间分析，发现程序运行的瓶颈。第 4 章介绍了利用 c-mex 进行 CUDA 编程的方法。第 5 章和第 6 章分别介绍了 MATLAB 并行计算工具箱和 CUDA 加速函数库的使用方法。第 7 章以计算机图形学中的 Marching Cubes 算法为例，详细介绍了 GPU 的开发方法。第 8 章介绍了如何将 MATLAB 程序转换为 CUDA 程序。最后这两章是作者多年来实际开发的经验之谈。

在本书即将出版之际，要感谢杨雪莲、刘玉龙、冯如、白璐等几位同学在本书翻译和校对中所做工作。本书还得到了中央高校基本科研业务费项目（2014JBZ021）、轨道交通控制与安全国家重点实验室（北京交通大学）自主课题（RCS2014ZZ03）和中科院无线传感网与通信重点实验室开放课题（2013005）的资助。

GPU 作为研究热点，很多术语没有统一的中文译名，虽经过仔细推敲，但译者水平有限，书中难免存在不妥之处，敬请读者不吝赐教。

本书提供大量的源代码，有需要的读者可登录机械工业出版社的网站（www.cmpbook.com），免费注册并登录后进入“图书展示”页面，搜索到本书页面，点击“相关下载”即可下载本书源代码。

熊　磊

轨道交通控制与安全国家重点实验室（北京交通大学）

前　言

MATLAB 是广泛应用于快速原型设计和算法开发的仿真工具，功能强大，简单易用。许多实验室和研究机构都迫切地希望 MATLAB 代码能够更快地运行，以满足大运算量项目的需要。由于 MATLAB 采用向量/矩阵的数据形式，适合于并行处理，因此采用图形处理单元（Graphics Processing Unit，GPU）对提升 MATLAB 运行速度大有裨益。

本书主要面向工程、科学、技术等专业领域，需要利用 MATLAB 进行海量数据处理的师生和科研人员。MATLAB 用户可能来自各个领域，不一定都具有丰富的程序开发经验。对于那些没有程序开发基础的读者，利用 GPU 加速 MATLAB 需要对他们的算法进行移植，会引入一些不必要的麻烦，甚至还需要设定环境。本书面向具有一定或较多 MATLAB 编程经验，但对 C 语言和计算机并行架构不是很了解的读者，以帮助读者将精力集中在他们的研究工作上，从而避免因使用 GPU 和 CUDA 而对 MATLAB 程序而非算法本身进行大量调整。

作为入门读物，本书从基础知识开始，首先介绍如何设置 MATLAB 运行 CUDA（在 Windows 和 Mac OSX），创建 c-mex 和 m 文件；接着引导读者进入专业级别的主题，如第三方 CUDA 库。本书还提供了许多修改用户 MATLAB 代码的实用方法，以更好地利用 GPU 强大的计算能力。

本书将指导读者使用 NVIDIA 的 GPU 显著提升 MATLAB 的运行速度。NVIDIA 的 CUDA 作为一种并行计算架构，最早用于计算机游戏设计，但由于其高效的大规模计算能力，在基础科学和工程领域也声誉日隆。通过本书，读者无需付出很多的精力和时间，就可以利用 GPU 的并行处理和丰富的 CUDA 科学库，实现 MATLAB 代码的加速，从而提升读者的科研工作水平。

本书第 5 章将使用 Mathworks 并行计算工具箱。虽然 Mathworks 并行计算工具箱是提升 MATLAB 速度的有效工具，但当前的版本在成为通用速度提升解决方案方面还是存在一定的局限，此外该工具箱还需要额外付费购买。特别是，由于并行计算工具箱的目标在于多核、多计算机和/或集群分布式计算，以及 GPU 处理，GPU 优化以提升用户代码运算速度，相对而言既受限于速度提升，又受限于所支持的 MATLAB 函数。此外，如果仅局限于 Mathworks 并行计算工具箱，就很难最大化利用丰富的 CDUA 库。本书第 5 章将介绍当前并行处理工具箱的功能与局限。实践证明，采用 c-mex 的 GPU 是普适性地提升速度的更好方法，而且在当前的环境中能够更为灵活地使用。

通过阅读本书，读者很快就能体会到 MATLAB 代码运行速度惊人的提升，而且通过使用开源 CUDA 资源，可以更好地进行科学研究。支持 Windows 和 Mac 操作系统也是本书的特点之一。

目　　录

译者序
前言
第 1 章　不使用 GPU 实现 MATLAB 加速……1
1.1　本章学习目标……1
1.2　向量化……1
1.2.1　元素运算……2
1.2.2　向量/矩阵运算……3
1.2.3　实用技巧……4
1.3　预分配……5
1.4　for-loop……6
1.5　考虑稀疏矩阵形式……7
1.6　其他技巧……9
1.6.1　尽量减少循环中的文件读/写……9
1.6.2　尽量减少动态改变路径和改变变量类型……9
1.6.3　在代码易读性和优化间保持平衡……9
1.7　实例……9
第 2 章　MATLAB 和 CUDA 配置……17
2.1　本章学习目标……17
2.2　配置 MATLAB 进行 c-mex 编程……17
2.2.1　备忘录……17
2.2.2　编译器的选择……18
2.3　使用 c-mex 实现“Hello，mex!”……21
2.4　MATLAB 中的 CUDA 配置……23
2.5　实例：使用 CUDA 实现简单的向量加法……25
2.6　图像卷积实例……31
2.6.1　MATLAB 中卷积运算……31
2.6.2　用编写的 c-mex 计算卷积……33
2.6.3　在编写的 c-mex 中利用 CUDA 计算卷积……35

2.6.4 简单的时间性能分析 …… 39
2.7 总结 …… 39
第 3 章 通过耗时分析进行最优规划 …… 41
3.1 本章学习目标 …… 41
3.2 分析 MATLAB 代码查找瓶颈 …… 41
3.2.1 分析器的使用方法 …… 41
3.2.2 针对多核 CPU 更精确的耗时分析 …… 44
3.3 CUDA 的 c-mex 代码分析 …… 46
3.3.1 利用 Visual Studio 进行 CUDA 分析 …… 46
3.3.2 利用 NVIDIA Visual Profiler 进行 CUDA 分析 …… 52
3.4 c-mex 调试器的环境设置 …… 57
第 4 章 利用 c-mex 进行 CUDA 编程 …… 64
4.1 本章学习目标 …… 64
4.2 c-mex 中的存储布局 …… 64
4.2.1 按列存储 …… 64
4.2.2 按行存储 …… 67
4.2.3 c-mex 中复数的存储布局 …… 68
4.3 逻辑编程模型 …… 70
4.3.1 逻辑分组 1 …… 72
4.3.2 逻辑分组 2 …… 73
4.3.3 逻辑分组 3 …… 73
4.4 GPU 简单介绍 …… 74
4.4.1 数据并行 …… 74
4.4.2 流处理器 …… 74
4.4.3 流处理器簇 …… 74
4.4.4 线程束 …… 75
4.4.5 存储器 …… 77
4.5 第一种初级方法的分析 …… 77
4.5.1 优化方案 A：线程块 …… 79
4.5.2 优化方案 B …… 84
4.5.3 总结 …… 86
第 5 章 MATLAB 与并行计算工具箱 …… 87
5.1 本章学习目标 …… 87

5.2 GPU 处理 MATLAB 内置函数 …… 87
5.3 GPU 处理非内置 MATLAB 函数 …… 93
5.4 并行任务处理 …… 95
5.4.1 MATLAB worker …… 95
5.4.2 parfor …… 97
5.5 并行数据处理 …… 99
5.5.1 spmd …… 99
5.5.2 分布式数组与同分布数组 …… 101
5.5.3 多个 GPU 时的 worker …… 105
5.6 无需 c-mex 的 CUDA 文件直接使用 …… 105
第 6 章 使用 CUDA 加速函数库 …… 111
6.1 本章学习目标 …… 111
6.2 CUBLAS …… 111
6.2.1 CUBLAS 函数 …… 112
6.2.2 CUBLAS 矩阵乘法 …… 113
6.2.3 使用 Visual Profiler 进行 CUBLAS 分析 …… 120
6.3 CUFFT …… 122
6.3.1 通过 CUFFT 进行二维 FFT 运算 …… 123
6.3.2 用 Visual Profiler 进行 CUFFT 时间分析 …… 130
6.4 Thrust …… 132
6.4.1 通过 Thrust 排序 …… 132
6.4.2 采用 Visual Profiler 分析 Thrust …… 134
第 7 章 计算机图形学实例 …… 136
7.1 本章学习目标 …… 136
7.2 Marching-Cubes 算法 …… 136
7.3 MATLAB 实现 …… 139
7.3.1 步骤 1 …… 139
7.3.2 步骤 2 …… 140
7.3.3 步骤 3 …… 141
7.3.4 步骤 4 …… 141
7.3.5 步骤 5 …… 142
7.3.6 步骤 6 …… 142
7.3.7 步骤 7 …… 143

7.3.8 步骤 8 …… 144
7.3.9 步骤 9 …… 145
7.3.10 时间分析 …… 151
7.4 采用 CUDA 和 c-mex 实现算法 …… 152
7.4.1 步骤 1 …… 152
7.4.2 步骤 2 …… 155
7.4.3 时间分析 …… 156
7.5 用 c-mex 函数和 GPU 实现 …… 157
7.5.1 步骤 1 …… 157
7.5.2 步骤 2 …… 158
7.5.3 步骤 3 …… 159
7.5.4 步骤 4 …… 164
7.5.5 步骤 5 …… 165
7.5.6 时间分析 …… 166
7.6 总结 …… 166
第 8 章 CUDA 转换实例：3D 图像处理 …… 168
8.1 本章学习目标 …… 168
8.2 基于 Atlas 分割方法的 MATLAB 代码 …… 168
8.2.1 基于 Atlas 分割背景知识 …… 168
8.2.2 用于分割的 MATLAB 代码 …… 169
8.3 通过分析进行 CUDA 最优设计 …… 177
8.3.1 分析 MATLAB 代码 …… 177
8.3.2 结果分析和 CUDA 最优设计 …… 181
8.4 CUDA 转换 1——正则化 …… 182
8.5 CUDA 转换 2——图像配准 …… 187
8.6 CUDA 转换结果 …… 200
8.7 结论 …… 202
附录 …… 203
附录 A 下载和安装 CUDA 库 …… 203
A.1 CUDA 工具箱下载 …… 203
A.2 安装 …… 203
A.3 确认 …… 206
附录 B 安装 NVIDIA Nsight 到 Visual Studio …… 207

第 1 章　不使用 GPU 实现 MATLAB 加速

1.1　本章学习目标

本章主要内容为 MATLAB 加速的基本方法，即不使用 GPU 和 c-mex 的固有方法。在本章中，你可以了解到以下内容：

- 采用向量化实现并行处理。
- 采用预分配实现内存有效管理。
- 其他加速 MATLAB 代码的有效方法。
- 循序渐进地提升代码性能的实例。

1.2　向量化

MATLAB 将数据表示为向量/矩阵的形式，所以“向量化”有助于加速 MATLAB 代码的运行。向量化的关键在于尽量减少 for 循环的使用。

考虑以下两个具有相同功能的 .m 文件：

```
% nonVec1.m

clear all;
tic

A = 0:0.000001:10;
B = 0:0.000001:10;

Z = zeros(size(A));
y = 0;

for i = 1:10000001

    Z(i) = sin(0.5*A(i)) * exp(B(i)^2);
    y = y + Z(i);

end
toc
y
```

```
% Vec1.m

clear all;
tic

A = 0:0.000001:10;
B = 0:0.000001:10;

Z = zeros(size(A));
y = 0;

y = sin(0.5*A) * exp(B.^2)';

toc
y
```

左侧的 nonVec1.m 文件使用 for 循环求和，而右侧的 Vec1.m 文件则没有。

```
>> nonVec1
Elapsed time is 0.944395 seconds.

y =
  -1.3042e+48

>> Vec1
Elapsed time is 0.330786 seconds.

y =
  -1.3042e+48
```

两个程序计算结果相同，但程序 Vec1.m 的计算时间大约为程序 nonVec1.m 的三分之一。所以为了更好地实现向量化，在代码中应尽量使用元素运算或者向量/矩阵运算。

1.2.1 元素运算

当用于两个矩阵时，符号*表示矩阵乘法，而符号 .* 则表示矩阵中对应元素相乘。例如，令 x=[1 2 3]，v=[4 5 6]：

```
>> k = x .* v
k =
   4  10  18
```

还有许多其他的运算也可以按元素进行：

```
>> k = x .^2
k =
   1 4 9

>> k = x ./ v
k =
   0.2500  0.4000  0.5000
```

很多函数也支持按元素运算：

```
>> k = sqrt(x)
k =
   1.0000  1.4142  1.7321

>> k = sin(x)
k =
   0.8415  0.9093  0.1411

>> k = log(x)
k =
   0  0.6931  1.0986

>> k = abs(x)
k =
   1  2  3
```

甚至关系运算符也可以按元素运算：

```
>> R = rand(2,3)
R =
   0.8147  0.1270  0.6324
   0.9058  0.9134  0.0975

>> (R > 0.2) & (R < 0.8)

ans =
     0  0  1
     0  0  0

>> x = 5

x =
   5

>> x >= [1 2 3; 4 5 6; 7 8 9]

ans =
     1  1  1
     1  1  0
     0  0  0
```

甚至更复杂的组合运算也可以按元素进行：

```
>> A = 1:10;
>> B = 2:11;
>> C = 0.1:0.1:1;
>> D = 5:14;

>> M = B ./ (A .* D .* sin(C));
```

1.2.2　向量/矩阵运算

MATLAB 基于线性代数软件包，而在线性代数中使用向量/矩阵运算能够有效地替代 for 循环，提高运算速度。矩阵乘法是最常见的向量/矩阵运算，该运算是对每个元素进行乘法和加法的组合运算。

先考虑两个列向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$，那么它们的点积为 1×1 的矩阵，如下所示：

$$\boldsymbol{a}=\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix},\ \boldsymbol{b}=\begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}$$

$$\boldsymbol{a}\cdot\boldsymbol{b}=\boldsymbol{a}^{\mathrm{T}}\boldsymbol{b}=\begin{bmatrix} a_x & a_y & a_z \end{bmatrix}\begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}=\begin{bmatrix} a_xb_x+a_yb_y+a_zb_z \end{bmatrix}$$

如果两个向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 均是行向量，那么 $\boldsymbol{a}\cdot\boldsymbol{b}$ 计算定义为 $\boldsymbol{a}\boldsymbol{b}^{\mathrm{T}}$，由乘法和加法的组合运算，得到 1×1 的矩阵，如下：

```
A = 1:10  % 1×10 matrix
B = 0.1:0.1:1.0  % 1×10 matrix

C = 0;

for i = 1:10

    C = C + A(i) * B(i);

end
```

```
A = 1:10  % 1×10 matrix
B = 0.1:0.1:1.0  % 1×10 matrix

C = 0;

C = A*B';  % A·B^T
```

在许多情况下，以向量运算的形式考虑矩阵乘法是很有效的。例如，可以将矩阵—向量乘法 $\boldsymbol{y}=\boldsymbol{Ax}$ 分解为 $\boldsymbol{x}$ 和矩阵 $\boldsymbol{A}$ 各行的点乘：

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_i \end{bmatrix} = \begin{bmatrix} \boldsymbol{a}_1 \\ \boldsymbol{a}_2 \\ \vdots \\ \boldsymbol{a}_i \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \end{bmatrix}$$

$$y_i = \boldsymbol{a}_i \cdot \boldsymbol{x}$$

1.2.3 实用技巧

在许多应用中，需要对每个元素设定上边界和下边界。为了实现这一目的，通常使用 if 和 elseif 语句，但这容易破坏向量化。因此，可以使用内置函数 min 和 max 替代 if 和 elseif 语句设置元素边界：

```
% ifExample.m

clear all;
tic

A = 0:0.000001:10;
B = 0:0.000001:10;

Z = zeros(size(A));
y = 0;

for i = 1:10000001

  if(A(i) < 0.1) A(i) = 0.1;
  elseif(A(i)> 0.9) A(i) = 0.9;
  end

  Z(i) = sin(0.5*A(i)) * exp(B(i)^2);
  y = y + Z(i);

end

toc
y
```

```
% nonifExample.m

clear all;
tic

A = 0:0.000001:10;
B = 0:0.000001:10;

Z = zeros(size(A));
y = 0;

A = max(A, 0.1);
% max(A, LowerBound)
% A >= LowerBound

A = min(A, 0.9);
% min(A, UpperBound)
% A <= UpperBound

y = sin(0.5*A) * exp(B.^2)';

toc
y
```

```
>> ifExample
Elapsed time is 0.878781 seconds.

y =
   5.8759e + 47

>> nonifExample
Elapsed time is 0.309516 seconds.

y =
   5.8759e + 47
```

同样地，如果需要查找和替换某些元素的值，可以用 find 函数替代 if 和 elseif 来保持向量化：

```
% ifExample2.m

clear all;
tic

A = 0:0.000001:10;
B = 0:0.000001:10;

Z = zeros(size(A));
y = 0;

for i = 1:10000001

if(A(i) == 0.5) A(i) = 0;
end

   Z(i) = sin(0.5*A(i)) * exp(B(i)^2);
   y = y + Z(i);

end

toc
y
```

```
% nonifExample2.m

clear all;
tic

A = 0:0.000001:10;
B = 0:0.000001:10;

Z = zeros(size(A));
y = 0;

% Vector A is compared with scalar
% 0.5

A(find(A == 0.5)) = 0;

y = sin(0.5*A) * exp(B.^2)';

toc
y
```

将向量 $\boldsymbol{A}$ 中的元素与标量 0.5 进行比较，返回一个与向量 $\boldsymbol{A}$ 相同大小的向量，$\boldsymbol{A}$ 中元素为 0.5 的位置设为 0。find 函数能够给出匹配位置的索引，并替换初始值。

1.3　预分配

由于每次调整数组的大小都涉及内存的释放或分配，以及数值的复制，极为耗费时间。所以通过为所需数组预分配内存，能够获得相当显著的加速。

```
% preAlloc.m

% Resizing Array
tic

x = 8;
```

```
x(2) = 10;
x(3) = 11;
x(4) = 20;

toc

% Pre-allocation
tic

y = zeros(4,1);
y(1) = 8;
y(2) = 10;
y(3) = 11;
y(4) = 20;

toc
>> preAlloc
Elapsed time is 0.000032 seconds.
Elapsed time is 0.000014 seconds.
```

在上面的例子中，多次调整了数组 x 的大小，需要对内存重新分配和设置数值，而数组 y 则通过 zeros(4, 1) 进行了初始化。如果在运算前不知道矩阵的大小，可以预先设置一个足够大的数组来存储数据，这样使用该数组进行运算时就不需要进行内存的重新分配了。

1.4 for-loop

许多情况下，不可避免地需要使用 for-loop。作为 Fortran 的演进，MATLAB 按列顺序存储矩阵，即第一列的元素按顺序存储在一起，然后是第二列的元素，以此类推。由于内存为线性结构，系统能够缓存附近元素的值，所以对矩阵按列嵌套循环更有利于程序运行。

```
% row_first.m

clear all;

A = rand(300,300,40,40);
B = zeros(300,300,40,40);

tic
for i = 1:300
  for j = 1:300

    B(i,j,:,:) = 2.5 * A(i,j,:,:);

  end
end

toc
```

```
% col_first.m

clear all;

A = rand(300,300,40,40);
B = zeros(300,300,40,40);

tic
for j = 1:300
  for i = 1:300

    B(i,j,:,:) = 2.5 * A(i,j,:,:);

  end
end

toc
```

```
>> row_first
Elapsed time is 9.972601 seconds.
>> col_first
Elapsed time is 7.140390 seconds.
```

上述例子中，通过嵌套 for-loop 中 i 和 j 的简单对换，col_first.m 的运行速度比 row_first.m 快大约 30%。所以当使用大规模高维矩阵时，进行以列为主的运算能够获得更高的速度增益。

1.5　考虑稀疏矩阵形式

对于稀疏矩阵（大部分元素是零的大规模矩阵）的处理，使用 MATLAB 中的“稀疏矩阵形式”是一种很好的想法。对于非常大的矩阵，因为只需存储非零元素，所以稀疏矩阵所需内存更少，而且运算过程中不考虑零元素，运算速度也更快。

创建稀疏形式矩阵的简易方法如下：

```
>> i = [5 3 6] % used for row index

>> j = [1 6 3] % used for column index

>> value = [0.1  2.3  3.1] % used for values to fill in

>> s = sparse (i,j,value) % generate sparse matrix

s =
   (5,1)    0.1000
   (6,3)    3.1000
   (3,6)    2.3000

>> full = full(s) % convert the sparse matrix to full matrix

full =
        0         0         0         0         0         0
        0         0         0         0         0         0
        0         0         0         0         0    2.3000
        0         0         0         0         0         0
   0.1000         0         0         0         0         0
        0         0    3.1000         0         0         0
```

或者简单地使用内置指令 sparse 将全矩阵转化为稀疏矩阵：

```
>> SP = sparse(full)

SP =
    (5,1)    0.1000
    (6,3)    3.1000
    (3,6)    2.3000
```

所有 MATLAB 内置运算都适用于稀疏矩阵形式，无需进行任何修改，而且稀疏矩阵的运算结果也是稀疏矩阵形式。

稀疏矩阵形式的运行效率由矩阵的稀疏度决定，即矩阵中零元素越多，矩阵运算速度越快。如果一个矩阵几乎不稀疏，那么使用稀疏矩阵形式，将不会带来速度的提升和内存的节省。

```
% DenseSparse.m
% Dense matrix
A = rand (5000,2000);

b = rand (5000,1);
tic
   x = A\b ;
toc
% Convert to sparse form in MATLAB
sp_denseA = sparse(A);
tic
   x = sp_denseA\b ;
toc
```

```
% RealSparse.m
% Sparse matrix
sp_A2 = sprand(5000, 2000, 0.2);
b2 = rand (5000,1);
% Convert sparse matrix to full
% matrix
full_A2 = full(sp_A2);
tic
   y = full_A2\b ;
toc
tic
   y = sp_A2\b2 ;
toc
```

```
>> denseSparse
Elapsed time is 5.846979 seconds.
Elapsed time is 41.050271 seconds.

>> RealSparse
Elapsed time is 5.879175 seconds.
Elapsed time is 3.798073 seconds.
```

在例子 DenseSparse.m 中，尝试将密集矩阵（A:5000×5000）转换为稀疏矩阵（sp_denseA），稀疏矩阵形式的运算比密集矩阵花费了更多时间。然而在例子 RealSparse.m 中，稀疏矩阵形式的速度提高了很多。函数 sprand(5000, 2000, 0.2) 用于生成 5000×2000 的稀疏矩阵。第三个参数 0.2 意味着，在生成的矩阵中大约 20%的元素为非零元素，其他 80%的元素为零元素。在矩阵运算中，即使有 20%的非零元素，稀疏矩阵也能够获得非常大的速度提升。

观察每个矩阵所需的内存大小（如下所示），稀疏形式的稀疏矩阵（sp_A2）比全矩阵（80MB）需要更少的内存（大约 29MB）。然而稀疏形式的密集矩阵（sp sp_denseA）比全矩阵（80MB）需要更多的内存（大约 160MB），这是因为稀疏矩阵形式需要更多的内存来存储矩阵的索引信息。

```
>> whos('A','sp_denseA','full_A2','sp_A2')
Name              Size          Bytes      Class    Attributes

  A             5000×2000     80000000    double
full_A2         5000×2000     80000000    double
sp_A2           5000×2000     29016504    double     sparse
sp_denseA       5000×2000    160016008    double     sparse
```

1.6 其他技巧

1.6.1 尽量减少循环中的文件读/写

当调用的函数涉及文件读/写时，是十分花费时间的，因此，需要尽量减少，甚至避免调用函数，特别是在循环中。处理文件读/写最好的方式是在循环开始前读取文件，并将数据存储于变量中；然后在循环中使用这些存储的变量；在退出循环后，将结果重新写入文件。

虽然避免在循环中进行文件读/写看起来十分浅显，但是当在多个文件中使用多个函数时，文件读/写会不经意间驻留在深层的内循环中。为了检测这一无心之失，推荐使用分析器来分析代码中的哪一部分显著影响了总的处理时间。第 3 章将介绍分析器的使用。

1.6.2 尽量减少动态改变路径和改变变量类型

与文件读/写相同，设置路径也十分花费时间，应尽量减少使用。同样地，建议在循环外设置路径。改变变量、向量、矩阵的类型也十分花费时间，也应尽量减少，尤其是在循环中。

1.6.3 在代码易读性和优化间保持平衡

尽管本书一直在强调代码的优化，但是代码的可维护性也是算法设计/原型中的重要因素。因此，过于注重代码优化，而牺牲代码可读性，并不是一个好主意。应该力争在代码易读性和优化间保持平衡，这样才能从调试时间的节省中获得更大益处，从而将更多的时间用于创造性算法和其他事情。

1.7 实例

在运行 myDisplay.m 时，能看到如图 1.1 所示的结果。此例的目的是在图 1.1b 的 12 个目标中，找出一本黑色的书（左图为样板图像）。样板图像和目标图像均为彩色，均为三个维度（x，y，颜色）。尽管图 1.1b 中目标图像的原始大小为 1536×2048×3，图 1.1a 样板图像的大小为 463×463×3，但出于演示的目的，图 1.1b 图像的每个目标都已经被挑出，且具有与样板图像相同的大小（见图 1.2），以减少不必

要的细节。并且 12 个目标图像均以“展开”形式储存在 compareImageVector.mat 中的大矩阵 **v** 中，即每个目标图像大小为 463×463×3 的矩阵被展开为大小为 1×643107 的行向量，所以对于 12 个目标，矩阵 **v** 共有 12 行（矩阵大小为 12×643107）。

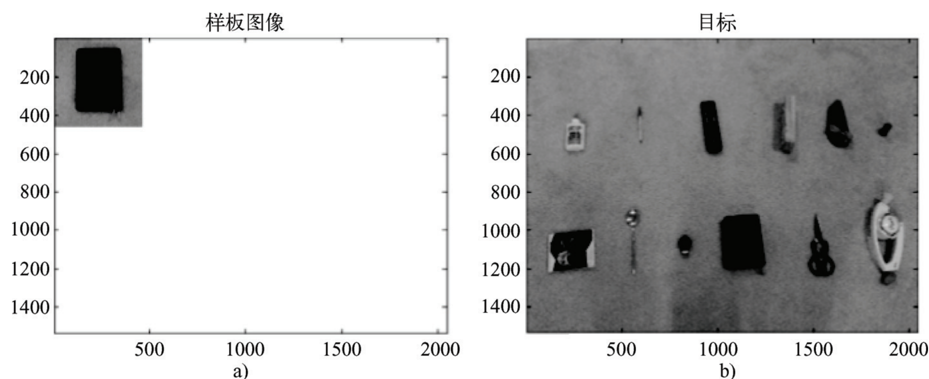


图 1.1　使用样板图像进行目标检测实例

图 1.1 中，将 a 图中黑色的书作为样板图像检测 a 图的 12 个目标。注意 b 图中黑色的书有轻微旋转，而且背景中有阴影存在

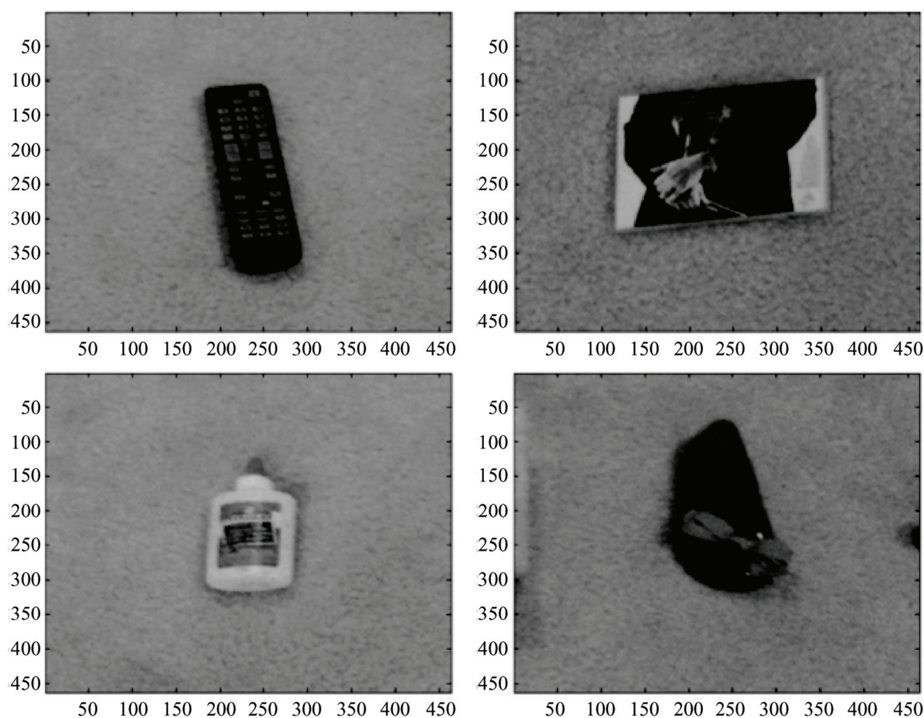


图 1.2　为简便起见，每个目标都被挑出，且具有与样板图像相同的大小

最初的 m 代码如下：

```
% compareVec_LoopNaive.m

load('compareImageVector.mat');

[sizeX, sizeY, sizeZ] = size(template);

tic

mean_v = mean(v')';
std_v = std(v')';

mean_template = mean(template_vec);
std_template = std(template_vec);

for i = 1:12
  y(i,1) = 0;
  for j = 1:size(v,2)

     normalized_v(i,j) = (v(i,j) - mean_v(i));
     normalized_template(j,1) = template_vec(j,1) - mean_template;
     y(i) = y(i) + (normalized_v(i,j) * normalized_template(j,1))/ ...
          (std_v(i,1)*std_template);

  end
end

toc
y
```

此 m 文件可计算用于比较各目标图像与样板图像的归一化互相关值，计算方法如下：

$$\sum_{x,\ y}\frac{\left[I(x,y)-\overline{I}\right]\left[T(x,y)-\overline{T}\right]}{\sigma_I\sigma_T}$$

式中，$T(x,\ y)$是样板图像中每个像素的值；$\overline{T}$ 表示样板图像的平均像素值；σ_T 是标准差；$I(x,\ y)$是进行比较的目标图像中的各个像素值。所以，当计算出的归一化互相关值越大时，意味着相互比较的两个图像越相似。本例中，我们尝试找出归一化相关值最大的目标，即认为它与样板图像最相似。

compareVec_LoopNaive.m 运行结果如下：

```
>> compareVec_LoopNaive
Elapsed time is 65.008446 seconds.

y =
   1.0e + 05 *

   0.2002
```

```
 1.2613
 2.9765
 2.6793
 2.6650
-0.0070
 2.7570
 0.7832
 0.3291
 5.0278
 2.8071
-0.5271
```

由结果可知，第 10 行的归一化互相关值最大（5.0278×10^5），对应图 1.1 中的第 10 个目标。尽管上述代码能够找出正确的目标（见图 1.3），但是仍需要进行一些优化来提高代码的速度。在阅读下列代码之前，请先自行试着找出 compareVec_LoopNaive.m 中能够改进的部分。

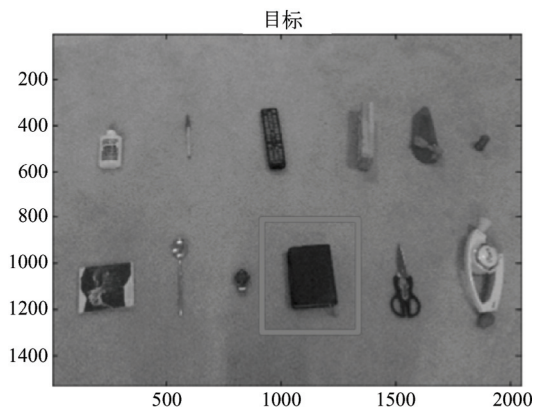


图 1.3　采用归一化互相关进行的各目标图像与样板图像的比较结果。具有最大归一化互相关值的第 10 个目标被选出

接下来仔细观察上述代码。你还记得预分配能够加速 MATLAB 吗？在使用变量 y、normalized_template 和 normalized_v 前，先通过 zero()函数进行内存分配（见 compareVec_Loop.m）。

```
% compareVec_Loop.m
load('compareImageVector.mat');
[sizeX, sizeY, sizeZ] = size(template);
y = zeros(12,1,'double');
tic
```

```
mean_v = mean(v')';
std_v = std(v')';

mean_template = mean(template_vec);
std_template = std(template_vec);

normalized_template = zeros(size(template_vec));
normalized_v = zeros(size(v));

for i = 1:12
  for j = 1:size(v,2)

     normalized_v(i,j) = (v(i,j) - mean_v(i));
     normalized_template(j,1) = template_vec(j,1) - mean_template;
     y(i) = y(i) + (normalized_v(i,j) * normalized_template(j,1))/ ...
         (std_v(i,1)*std_template);

  end
end

toc
y
```

结果如下：

```
>> compareVec_Loop
Elapsed time is 58.102506 seconds.

y =
   1.0e + 05 *

   0.2002
   1.2613
   2.9765
   2.6793
   2.6650
  -0.0070
   2.7570
   0.7832
   0.3291
   5.0278
   2.8071
  -0.5271
```

通过简单的内存预分配，毫不费力地节省了大量的运行时间（从 65s 减少到 58s）。下面再调整循环中的第一步运算。

由于 normalized_v(i, j)=(v(i, j)-mean_v(i)) 是各行简单的减法运算，所以可以通过向量运算将这行运算放到循环外：

```
mean_matrix = mean_v(:,ones(1,size(v,2)));
normalized_v = v - mean_matrix;
```

这里，通过重复 mean_v 使得 mean_matirx 具有与矩阵 v 相同的大小。通过这个简单的改变，能够减少运行时间。

```
% compareVec_LoopImprove.m

load('compareImageVector.mat');

[sizeX, sizeY, sizeZ] = size(template);
y = zeros(12,1,'double');

tic

mean_v = mean(v')';
std_v = std(v')';

mean_template = mean(template_vec);
std_template = std(template_vec);

normalized_template = zeros(size(template_vec));
normalized_v = zeros(size(v));

mean_matrix = mean_v(:,ones(1,size(v,2)));
normalized_v = v - mean_matrix;

for i = 1:12
  for j = 1:size(v,2)

    normalized_template(j,1) = template_vec(j,1) - mean_template;
    y(i) = y(i) + (normalized_v(i,j) * normalized_template(j,1))/ ...
        (std_v(i,1)*std_template);

  end
end

toc
y
>> compareVec_LoopImprove
Elapsed time is 51.289607 seconds.

y =
   1.0e + 05 *

   0.2002
   1.2613
   2.9765
   2.6793
   2.6650
  -0.0070
   2.7570
   0.7832
   0.3291
   5.0278
   2.8071
  -0.5271
```

下面继续优化代码。可以使用元素运算和向量/矩阵运算来避免使用 for-loop。对应元素运算，可以进行如下修改：

```
mean_matrix = mean_v(:,ones(1,size(v,2)));
normalized_v = v - mean_matrix;

mean_template = mean(template_vec);
std_template = std(template_vec);
normalized_template = template_vec - mean_template;
```

这些语句能够有效地替代 for-loop。

而进一步采用向量/矩阵运算，能够完全不使用 for-loop：

```
y = normalized_v * normalized_template;
```

最终的向量化 m-code 如下：

```
% compareVec.m

load('compareImageVector.mat');

[sizeX, sizeY, sizeZ] = size(template);
y = zeros(12,1,'double');

tic

mean_v = mean(v')';
std_v = std(v')';

mean_matrix = mean_v(:,ones(1,size(v,2)));
normalized_v = v - mean_matrix;

mean_template = mean(template_vec);
std_template = std(template_vec);
normalized_template = template_vec - mean_template;

y = normalized_v * normalized_template;
y = y./(std_v*std_template);

toc
y
```

运行结果如下：

```
>> compareVec
Elapsed time is 0.412896 seconds.
y =
   1.0e + 05 *

   0.2002
   1.2613
   2.9765
   2.6793
   2.6650
```

```
-0.0070
 2.7570
 0.7832
 0.3291
 5.0278
 2.8071
-0.5271
```

由结果可知，优化后的代码比使用两重循环嵌套和未进行预分配的原始代码要快得多（在不损失准确度的情况下，运行时间由 65s 降为 0.41s）。

第 2 章　MATLAB 和 CUDA 配置

2.1　本章学习目标

采用 C/C++语言编写的 MATLAB 函数称为 c-mex 文件。c-mex 文件是在 MATLAB 中使用 CUDA 和 GPU 的基本出发点。在 MATLAB 会话中，这些 c-mex 函数将动态加载为函数。编写自己的 c-mex 文件有很多原因：

1）在 MATLAB 中重用 C/C++函数。

2）提高运行速度。

3）无限的自定义扩展。

尽管 Mathworks 的并行计算工具包和其他第三方 CPU 工具包提供了 CUDA 接口，但是要充分利用 CUDA 和 GPU 仍存在很多约束和限制。而使用 c-mex 文件这种通用方法，能够提供无止境的自定义扩展，并且在使用 NVIDIA 和其他公司提供的 CUDA 库时十分灵活。在本章中，可以了解到以下内容：

- c-mex 编程的 MATLAB 配置。
- 编写最简单的 c-mex 实例——“Hello，c-mex”。
- MATLAB 中的 CUDA 配置。
- 用 MATLAB 编写 CUDA 简单实例。

2.2　配置 MATLAB 进行 c-mex 编程

2.2.1　备忘录

MATLAB Executable（MEX）用于在 MATLAB 环境中直接使用 C/C++和 FORTRAN 代码，以实现更高的运算速度和避免应用瓶颈。我们将 C/C++ MEX 称为 c-mex，并且本书为了达到有效利用 GPU 设备的目的，只关注 c-mex。由于 c-mex 需要创建 C/C++可执行代码，而 CUDA 需要针对硬件（NVIDIA GPU）的代码，所以除了标准的 MATLAB 安装，还需要额外的安装步骤。首

先检查 C/C++编译器的安装状况，然后进行 CUDA 的安装。

1. C/C++编译器

当进行 c-mex 编程时，MATLAB 利用安装在操作系统中的 C/C++编译器创建 MATLAB 可调用的二进制文件。所以你应该知道操作系统中哪个编译器可用，以及编译器的安装位置，并且确保 MATLAB 版本支持操作系统中的编译器。为此，你可能需要浏览 Mathworks 网站，检查安装 MATLAB 与编译器版本的兼容性，网址为 http://www.mathworks.com/support/compilers/R2013a /index.html。通常在 Windows 中，Microsoft Visual C++编译器 cl.exe 安装在以下位置[㊀]：

- C:\Program Files (x86)\Microsoft Visual Studio x.0\VC\bin 64 位操作系统
- C:\Program Files\Microsoft Visual Studio x.0\VC\bin 32 位操作系统

在 Mac OS X 和 Linux 发行套件中，MATLAB 支持 gcc/g++ 编译器，gcc/g++ 编译器的安装位置由 Linux 发行套件决定，一般安装在以下位置：

- /Developer/usr/bin Mac OS X
- /usr/local/bin Linux 发行版

你需要核实是否已正确安装编译器，如果编译器安装在其他位置，请记下安装位置。

2. NVIDIA CUDA 编译器 nvcc

要编译 CUDA 代码，你需要从 NVIDIA 网站下载并安装 CUDA 工具包。这个工具包可以免费获得。CUDA 工具包的下载、安装步骤和信息请参考附录 A。

nvcc 能够翻译 CUDA 专用代码，并调用 C/C++编译器生成可执行的二进制文件或目标文件。因此，同时需要 CUDA 编译器和 C/C++编译器才能创建 GPU 可执行文件。

需注意的是，我们有必要事先知道编译器的位置以及 CUDA 运行时库的位置。

很多时候，大多数的编译错误都源于错误定义或者不定义编译器和库的位置。一旦确定了它们的位置，并且在系统环境中相应地设置其路径，那么开始 c-mex 和 CUDA 编程将会变得非常容易和顺畅。

2.2.2 编译器的选择

首先在 MATLAB 中选择 C 编译器。在 MATLAB 命令窗口中，运行 mex -setup 命令。收到欢迎信息后按[y]键继续，令 mex 定位已经安装的编译器的位置，如图 2.1 所示。

㊀ 本书以 MATLAB 2013a 为例，使用其他版本的读者请浏览 Mathworks 网站上的相应网页，检查兼容性。

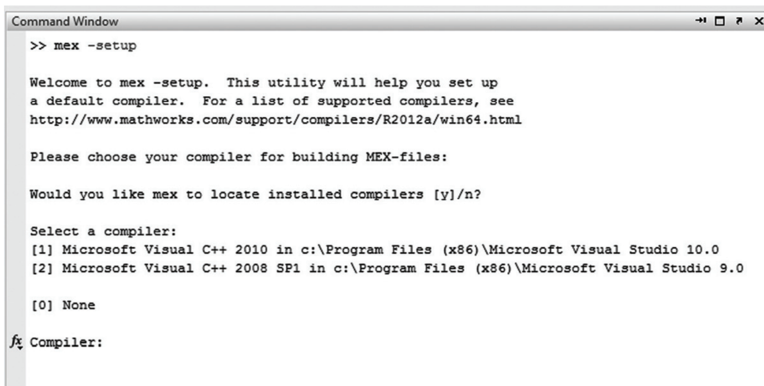


图 2.1　在 MATLAB 命令窗口中的 c-mex 配置信息

在本例中，有两个 Microsoft Visual C++编译器可用；选择[1]将 Microsoft Visual C++ 2010 作为 C++编译器。MATLAB 会询问是否确认选择，按[y]键确认。一旦 MATLAB 更新配置文件，就要完成编译器的选择。更新的 c-mex 配置文件包含使用的 C++编译器信息，以及如何编译和链接 C++代码。

配置文件实际上是由 mex 支持的模板生成和更新的。所有 mex 支持的模板配置文件在 Windows 系统中都位于 MATLABroot\bin\win32\mexopts（32 位操作系统）或 MATLABroot\bin\win64\mexopts（64 位操作系统）；在 UNIX 系统中，则位于 MATLABroot/bin 文件夹。图 2.2 展示了 MATLAB 命令窗口中 c-mex 设置的实例会话。

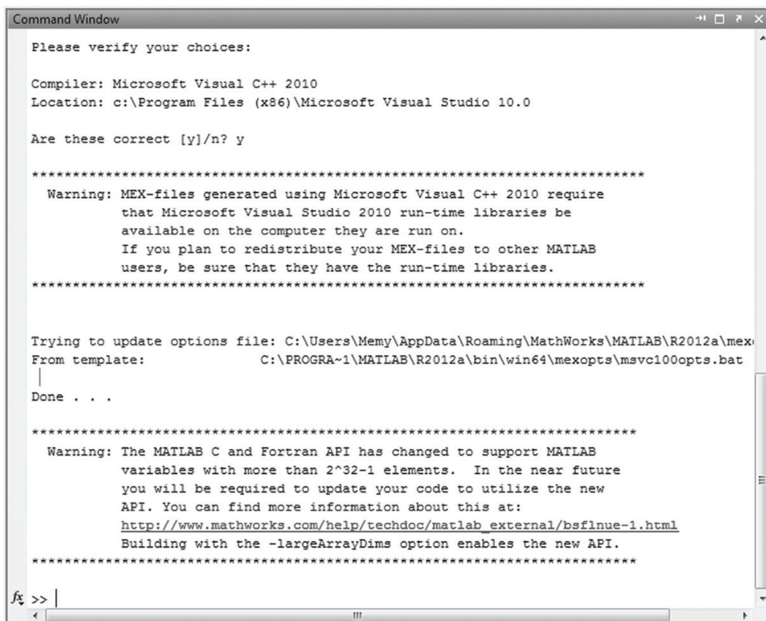


图 2.2　c-mex 配置中确认 C/C++编译器

对于选择的编译器，MATLAB 使用一个内置模板。MATLAB 会给出该 MATLAB 版本所支持编译器的列表。最终的配置文件将存储所有用于 c-mex 编译的编译器特定编译和链接选项。对于特殊的编译需求，可以编译这个配置文件，例如警告和调试选项。

可以在如下位置找到配置文件的本地副本：

● Windows 7 操作系统

C:\Users\MyName\AppData\Roaming\MathWorks\MATLAB\R2012a\mexopts.bat.

● Windows XP 操作系统

C:\Documents and Settings\MyName\Application Data\MathWorks\MATLAB \R2012a\mexopts.bat.

● Mac OS X 操作系统

/Users/MyName/.MATLAB/R2012a/mexopts.sh.

如果打开 Mac 中的配置文件，也可以指定希望在编译器中使用的 SDK。可以简单地编辑 SDKR00T 并保存（见图 2.3）。

● Linux 发行版

~/.MATLAB/R2012a/mexopts.sh.

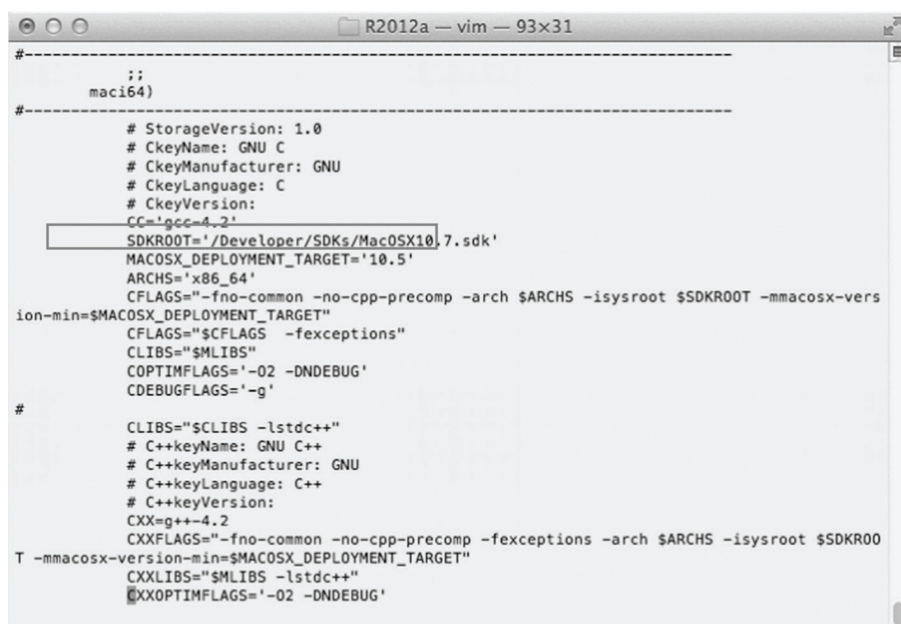


图 2.3　用于 Mac OS X SDK 选择的 mexopts.sh 文件

MATLAB 支持的编译器随操作系统和 MATLAB 版本的不同而不同。再次强调，必须确保安装的编译器能够为系统中已安装的 MATLAB 版本所支持。

2.3　使用 c-mex 实现“Hello，mex!”

现在循序渐进地用 c-mex 实现“Hello，mex!”。

步骤 1　创建一个空工作文件夹，例如，

c:\junk\MATLABMeetsCuda\Hello

步骤 2　打开 MATLAB，并在 MATLAB 工具栏中把工作路径设置为当前文件夹，如图 2.4 所示。

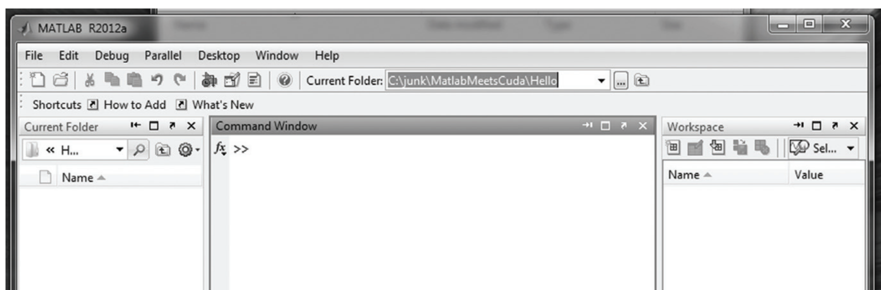


图 2.4　设置工作路径为当前文件夹

步骤 3　打开 MATLAB 编译器，在菜单中选择 File>New>Script 创建新的脚本。然后在编译器中将新脚本保存为 helloMex.cpp，如图 2.5 所示。

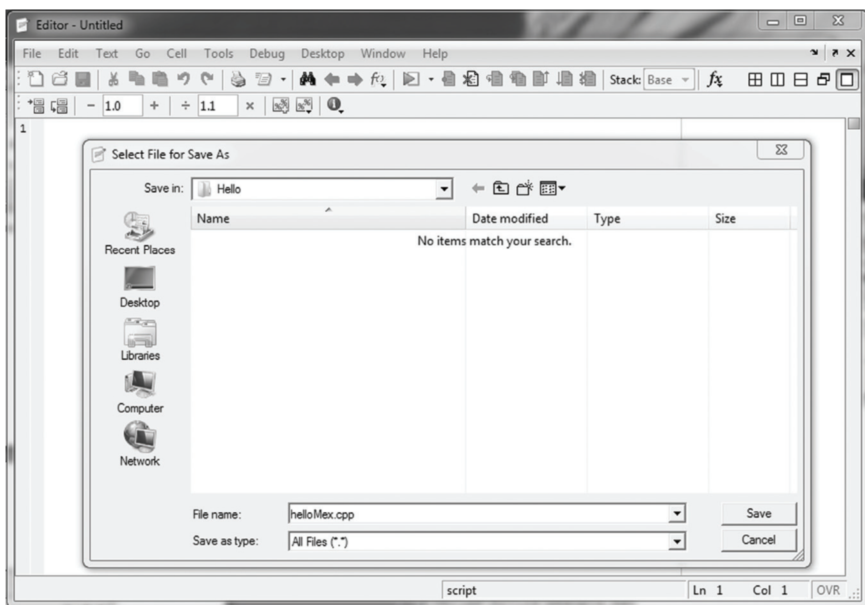


图 2.5　将新脚本保存为 C++文件

步骤 4 在编译窗中输入如下代码，选择菜单中的 File>Save 保存文件：

```
#include "mex.h"

void mexFunction(int nlhs,
                 mxArray *plhs[],

                 int nrhs,
                 const mxArray *prhs[])
{
    mexPrintf("Hello, mex!\n");
}
```

mexPrintf(..)与 C/C++中的 printf(..)等效。与在 printf 命令中打印 stdout 不同，mexPrintf 命令在 MATLAB 命令窗口中打印出编排好格式的指令信息。你能发现其使用方法与 printf 相同。

步骤 5 返回 MATLAB。MATLAB 在当前文件夹中显示新建文件 helloMex.cpp。然后在命令窗口运行以下指令，编译代码：

```
>> mex helloMex.cpp
```

c-mex 调用所选择的编译器完成编译、链接并最终生成二进制文件。生成的二进制文件能够为正常的 MATLAB 会话所调用。

步骤 6 上述步骤成功后，会在 Windows 系统相同文件夹下生成新的文件 helloMex.mexw64（或 helloMex.mexw32），如图 2.6 所示。

步骤 7 在命令窗口输入相应指令，就能向 c-mex 说“Hello”了，如图 2.7 所示。

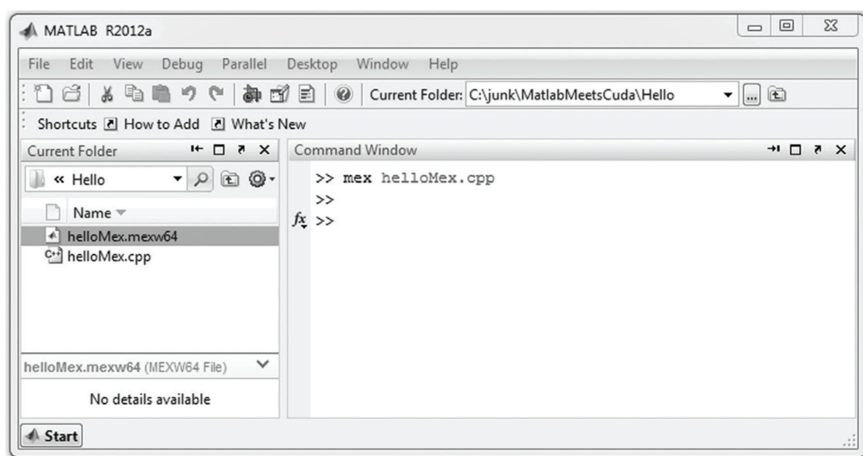


图 2.6 从 C/C++代码创建 c-mex 文件

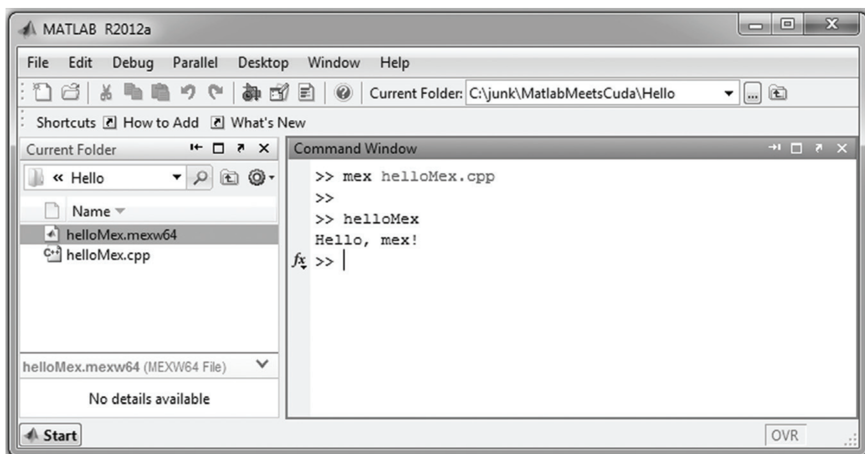


图 2.7　在命令窗口运行 HelloMex

```
>> helloMex
```

仅仅通过几行代码，我们就创建了第一个 c-mex 函数！

在上面的例子中，我们生成了一个特殊的函数 mexFunction。它一般称为子例行程序。这个函数提供了 mex 共享库的接入点，类似于 C/C++编程中的 main(…)函数。下面简单看看函数定义和它的输入参数：

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
mxArray *prhs[])

nlhs:  It is the number of output variables.
plhs:  Array of mxArray pointers to the output variables
nrhs:  Number of input variables
prhs:  Array of mxArray pointers to the input variables
```

在接下来的章节中可以了解更多细节。

2.4　MATLAB 中的 CUDA 配置

现在我们转向 CUDA。编译 CUDA 代码需要 nvcc 编译器。编译器能够翻译代码中 CUDA 专用代码，并且生成用于 GPU 和主机系统的机器代码。nvcc 在后台使用我们配置的 C/C++编译器。

在开始添加 CUDA 代码前，要先检查 CUDA 是否正确安装。CUDA Toolkit 默认安装在如下路径：

- Windows 操作系统

C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v#.#

其中，#.#是版本号（3.2 或者更高）。

- Mac OS X 操作系统

/Deverloper/NVIDIA/CUDA-#.#.

- Linux 操作系统

/usr/local/cuda-#.#.

如果安装正确，你应该可以在 MATLAB 控制界面使用如下指令：

```
>> system('nvcc')
```

如果运行这个语句，得到错误信息：

```
"nvcc : fatal error : No input files specified; use option -help for more information."
```

如信息所示，可以确定错误是由于没有指定要编译的输入文件。然而，如果 MATLAB 显示信息：

```
"nvcc is not recognized as an internal or external command, operable program or batch file"
```

则意味着 CUDA 没有正确安装。在进行下一步前，很可能需要参考 http://docs.nvidia.com/cuda/index.html 来确保在系统中正确安装 CUDA。

下面介绍一些确保 CUDA 环境正确配置的小技巧。

在 Windows 操作系统中，在命令提示符输入 path 指令检查路径：

```
C:\> path
```

在 PATH 变量中能看到 NVIDIA 编译器路径如下：

```
PATH=C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v#.#\bin\;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\Program Files\MATLAB\R2010b\bin;C:\Program Files\TortoiseSVN\bin;
```

或者打开 Control Panel＞All Control Panel Items＞System＞Advanced System Settings，如图 2.8 所示。关于 Windows 操作系统的更多信息参见 http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-microsoft-windows/index. html。

在 Mac OS X 操作系统中，在 Finder 中通过/Application/Utilities 找到终端应用。在 shell 中，输入以下代码：

```
imac:bin $ echo $PATH
/Developer/NVIDIA/CUDA-5.0/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/
local/bin:/usr/X11/bin
```

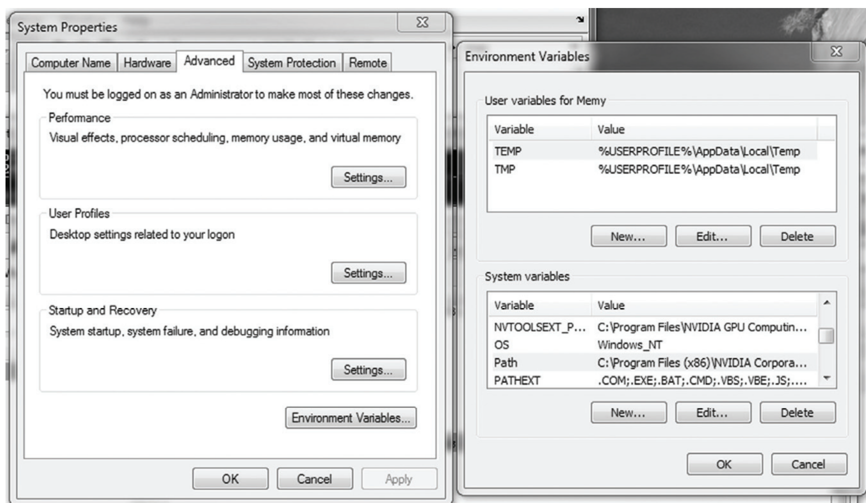


图 2.8　PATH 变量中 NVIDIA 编译器路径

你能在 CUDA 安装位置找到路径。更多关于 Mac OS X 操作系统的信息参见 http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-mac-osx/index.html。

在 Linux 中，例如，在 bash 中输入以下代码：

```
[user@linux_bash ~]$ echo $PATH
/usr/java/default/bin:     /opt/CollabNet_Subversion/bin:/bin:/sbin:/
usr:/usr/bin:/usr/sbin:/opt/openoffice.org3/program:/usr/local/cuda/
bin:/usr/java/default/bin:/usr/local/bin:/bin:/usr/bin:
```

然后寻找 nvcc 路径。在本例中，nvcc 安装在/usr/local/cuda/bin。更多 Linux 操作系统的信息参见 http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-linux/index.html。

2.5　实例：使用 CUDA 实现简单的向量加法

首先以简单通用的向量加法为例。本例需要创建一个 CUDA 函数，完成两个相同大小的输入向量的加法，并输出具有相同大小的独立向量作为结果。

步骤 1　在工作目录中创建 AddVectors.h，输入以下代码并保存：

```
#ifndef __ADDVECTORS_H__
#define __ADDVECTORS_H__

extern void addVectors(float* A, float* B, float* C, int size);

#endif // __ADDVECTORS_H__
```

在此头文件中，声明了 mex 函数中将要使用的向量加法函数原型。extern 表示该函数将在其他文件中执行。

步骤 2 在 AddVectors.cu 中实现 addVectors 函数。文件扩展名.cu 表示 CUDA 文件。在 MATLAB 编辑器中创建一个新文件。输入以下代码并保存为 AddVectors.cu：

```
#include "AddVectors.h"
#include "mex.h"

__global__ void addVectorsMask(float* A, float* B, float* C, int
size)
{
    int i = blockIdx.x;
    if (i >= size)
       return;

    C[i] = A[i] + B[i];
}

void addVectors(float* A, float* B, float* C, int size)
{
     float *devPtrA = 0, *devPtrB = 0, *devPtrC = 0;

     cudaMalloc(&devPtrA, sizeof(float) * size);
     cudaMalloc(&devPtrB, sizeof(float) * size);
     cudaMalloc(&devPtrC, sizeof(float) * size);

     cudaMemcpy(devPtrA, A, sizeof(float) * size,
     cudaMemcpyHostToDevice);
     cudaMemcpy(devPtrB, B, sizeof(float) * size,
     cudaMemcpyHostToDevice);

     addVectorsMask<<<size, 1>>>(devPtrA, devPtrB, devPtrC,
     size);

     cudaMemcpy(C, devPtrC, sizeof(float) * size,
     cudaMemcpyDeviceToHost);

     cudaFree(devPtrA);
     cudaFree(devPtrB);
     cudaFree(devPtrC);
}
```

步骤 3 此本步骤中，使用 -c 选项编译简单的 CUDA 代码，生成目标文件，该文件稍后将用于链接 mex 代码。由此代码建立目标文件，在 MATLAB 命令窗

口中输入以下指令：

```
>>system('nvcc -c AddVectors.cu')
```

成功后，你将会在命令窗口中看到 nvcc 返回如下所示相类似的信息：

```
AddVectors.cu
tmpxft_00000dc0_00000000-5_AddVectors.cudafe1.gpu
tmpxft_00000dc0_00000000-10_AddVectors.cudafe2.gpu
AddVectors.cu
tmpxft_00000dc0_00000000-5_AddVectors.cudafe1.cpp
tmpxft_00000dc0_00000000-15_AddVectors.ii
ans =
     0
```

如果命令窗口中显示如下错误信息

```
nvcc' is not recognized as an internal or external command, operable
program or batch file
```

这说明未在系统中设置 C++编译器路径。可以将 C++编译器路径添加到系统环境中，或者通过使用 -ccbin 选项明确设置：

```
>> system('nvcc -c AddVectors.cu -ccbin "C:\Program Files\Microsoft
Visual Studio 10.0\VC\bin"')
```

步骤 4　注意到在 MATLAB 当前文件夹窗口中，生成的目标文件位于相同的工作目录中，如图 2.9 所示。

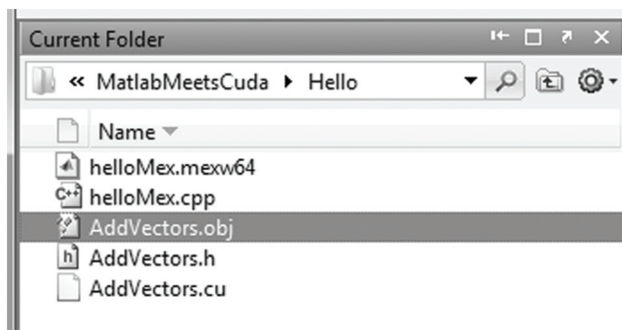


图 2.9　创建目标文件

步骤 5　创建 mex 函数，（也可称为 AddVectors 函数）。与 helloMex 函数一样，先创建 mexFunction。在 MATLAB 编辑器中创建新文件，输入以下代码，并保存为 AddVectorsCuda.cpp：

```
1   #include "mex.h"
2   #include "AddVectors.h"
3
4   void mexFunction(int nlhs, mxArray *plhs[], int nrhs, mxArray
    *prhs[])
5   {
6       if (nrhs != 2)
7           mexErrMsgTxt("Invaid number of input arguments");
8
9       if (nlhs != 1)
10          mexErrMsgTxt("Invalid number of outputs");
11
12      if (!mxIsSingle(prhs[0]) && !mxIsSingle(prhs[1]))
13          mexErrMsgTxt("input vector data type must be single");
14
15      int numRowsA = (int)mxGetM(prhs[0]);
16      int numColsA = (int)mxGetN(prhs[0]);
17      int numRowsB = (int)mxGetM(prhs[1]);
18      int numColsB = (int)mxGetN(prhs[1]);
19
20      if (numRowsA != numRowsB || numColsA != numColsB)
21          mexErrMsgTxt("Invalid size. The sizes of two vectors must
            be same");
22
23      int minSize = (numRowsA < numColsA) ? numRowsA : numColsA;
24      int maxSize = (numRowsA > numColsA) ? numRowsA : numColsA;
25
26      if (minSize != 1)
27          mexErrMsgTxt("Invalid size. The vector must be one
            dimentional");
28
29      float* A = (float*)mxGetData(prhs[0]);
30      float* B = (float*)mxGetData(prhs[1]);
31
32      plhs[0] = mxCreateNumericMatrix(numRowsA, numColsB,
        mxSINGLE_CLASS, mxREAL);
33      float* C = (float*)mxGetData(plhs[0]);
34
35      addVectors(A, B, C, maxSize);
36  }
```

第 6 行到第 13 行，是确保输入为支持的数据类型并修正向量大小。接着获取输入向量大小。第 32 行中，创建输出向量，存储两个向量相加的结果。第 35 行，调用基于 CUDA 的函数完成两个输入向量的相加。

步骤 6 编译 mex，并链接到创建的 CUDA 目标文件。在 MATLAB 命令窗口中输入以下命令。（运行环境为 Windows 64 位操作系统和 CUDA v5.0）：

```
>> mex AddVectorsCuda.cpp AddVectors.obj -lcudart -L"C:\Program
Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\lib\x64"
```

如果你安装的是 CUDA v4.0，将 v5.0 改为 v4.0。如果运行环境为 Windows 32 位操作系统，将 x64 改为 Win32。例如：

```
>> mex AddVectorsCuda.cpp AddVectors.obj -lcudart -L"C:\Program
Files\NVIDIA GPU Computing Toolkit\CUDA\v4.0\lib\Win32"
```

-lcudart 告知 mex 正在使用 CUDA 运行时库。

-L"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\lib\x64"告知 CUDA 运行时库的位置。

对于 MAC OS X 操作系统

```
>> mex AddVectorsCuda.cpp AddVectors.obj -lcudart -L"/Developer/
NVIDIA/CUDA-5.0/lib"
```

对于 Linux 发行套件

```
>> mex AddVectorsCuda.cpp AddVectors.obj -lcudart -L"/usr/local/
cuda/lib"
```

步骤 7　成功后，在同样的工作路径中会生成新的 mex 文件 AddVectorsCuda.mexw64，如图 2.10 所示。

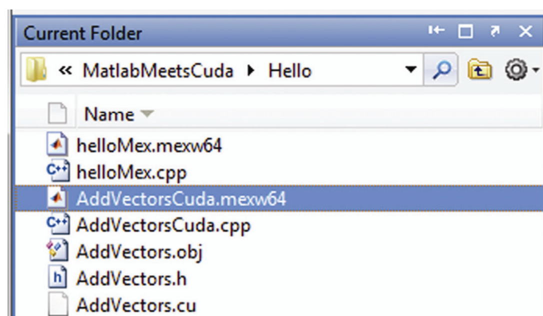


图 2.10　生成的 c-mex 文件

步骤 8　在 MATLAB 中运行新的 mex 函数。在命令窗口中，运行

```
>> A = single([1 2 3 4 5 6 7 8 9 10]);
>> B = single([10 9 8 7 6 5 4 3 2 1]);
>> C = AddVectorsCuda(A, B);
```

步骤 9　核实存储在向量 C 中的结果。当将每个向量元素相加时，结果向量 C 为 11：

```
>> C
C =
    11    11    11    11    11    11    11    11    11    11
```

可以通过 runAddVectors.m 运行整个过程，如下所示：

```
% runAddVectors.m

disp('1. nvcc AddVectors.cu compiling...');

system('nvcc -c AddVectors.cu -ccbin "C:\Program Files (x86)\Microsoft
Visual Studio 10.0\VC\bin"')

disp('nvcc compiling done !');

disp('2.  C/C++  compiling  for  AddVectorsCuda.cpp  with  AddVectors.
obj...');

mex  AddVectorsCuda.cpp  AddVectors.obj  -lcudart  -L"C:\Program  Files
\NVIDIA GPU Computing Toolkit\CUDA\v5.0\lib\x64"

disp('C/C++ compiling done !');

disp('3. Test AddVectorsCuda()...')

disp('Two input arrays:')
A = single([1 2 3 4 5 6 7 8 9 10])
B = single([10 9 8 7 6 5 4 3 2 1])

disp('Result:')
C = AddVectorsCuda(A, B)
```

综上所述，与 CUDA 相关的代码放在文件 AddVectors.cu 中。AddVectors.h 包含文件 AddVectors.cu 定义的函数原型。mex 函数（AddVectorsCuda.cpp 中的子例行程序）通过 AddVectors.h 调用 CUDA 函数。利用 nvcc.exe 将 CUDA 代码（.cu）编译为目标文件（.obj）后，使用 mex 命令编译 C/C++代码（.cpp）并将其链接到 CUDA 目标文件（.obj）。最终得到可执行的二进制 mex 文件（.mexw64），该文件包含有常规的 cpp 文件和 cu 文件。流程如图 2.11 所示。

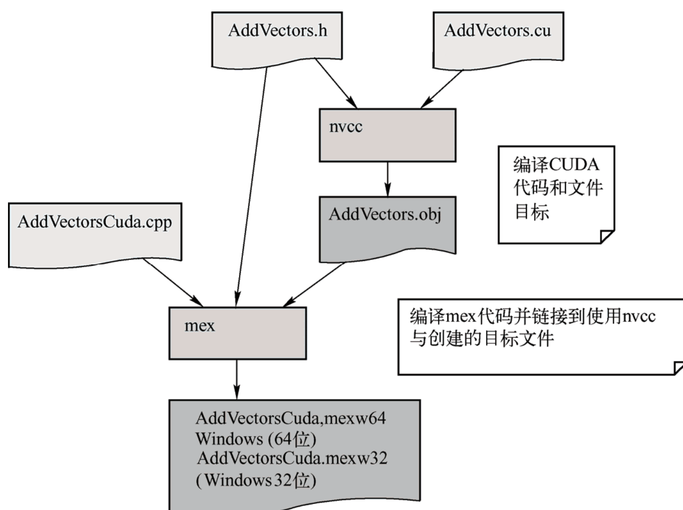


图 2.11　与 c-mex 编译相关的 CUDA 总结框图

（□：输入源文件，□：指令，□：生成文件）

2.6　图像卷积实例

现在创建一个更复杂的例子。首先，定义我们的 mex 函数。从 MATLAB 中读入一个样本图像，然后将其传送给 mex 函数，该函数接着使用 3×3 掩膜进行简单的二维卷积，结果返回 MATLAB。看看在下面三种情况下如何完成这个任务，这会十分有趣：

1）使用 MATLAB 函数 conv2。

2）使用纯 C++代码完成。

3）使用 CUDA 完成。

本节重点介绍在每种情况下如何用简单代码加以实现。在所有情况下，使用相同的图像（见图 2.12），它的数据类型是 single（单精度浮点数），与 3×3 的掩膜数据类型一样。实例中的掩膜如下：

$$\begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

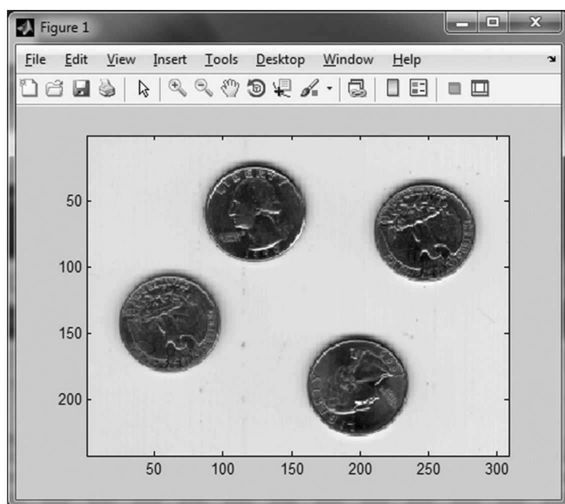


图 2.12　实例中的输入图像

2.6.1　MATLAB 中卷积运算

MATLAB 有内置二维卷积函数 conv2。conv2 使用方便，能够直接计算两个输入矩阵的二维卷积。首先看看在纯 MATLAB 上如何实现。

步骤 1　在 MATLAB 命令窗口中读入硬币的样本图像：

```
>> quarters = single(imread('eight.tif'));
>> mask = single([1 2 1; 0 0 0; -1 -2 -1]);
>> imagesc(quarters);
>> colormap(gray);
```

注意，输入图像和掩膜数据类型为 single，而当使用 imread 读取图像时，返回的图像数据类型为 uint8（8 位无符号整数）。由于 CUDA 中需要使用 single 数据类型，所以输入数据类型应转换为 single。

步骤 2 使用 conv2 进行二维卷积

```
>> H = conv2(quarters, mask, 'same');
```

从现在开始，卷积中的 shape 参数选择为‘same’。通过设定第三个参数为 same，表明要求 MATLAB 返回输出结果的大小与输入图像相同。现在，画出输出图像 H，观察梯度图像结果。

```
>> imagesc(H);
>> colormap(gray);
```

可以使用 convol_matlab.m 运行上述整个过程，代码如下：

```
% convol_matlab.m

quarters = single(imread('eight.tif'));
mask = single([1 2 1; 0 0 0; -1 -2 -1]);
imagesc(quarters);
colormap(gray);

H = conv2(quarters, mask, 'same');
imagesc(H);
colormap(gray);
```

结果图像如图 2.13 所示。

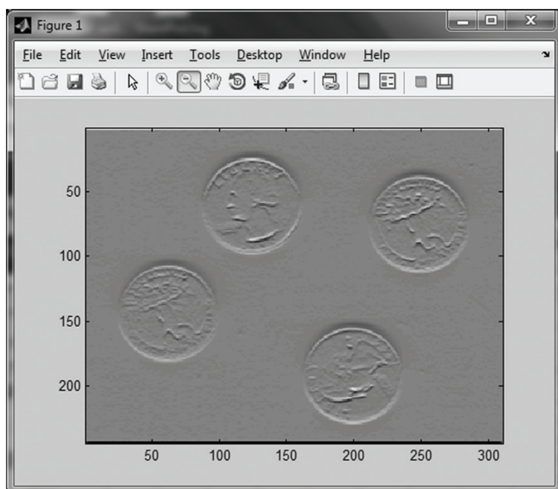


图 2.13 梯度图像结果

2.6.2　用编写的 c-mex 计算卷积

下面使用我们自己编写的 c-mex 函数执行与 conv2 函数相同的功能。在开始运行之前，先重温一下上个例子中介绍的子例行程序。子例行程序共有 4 个输入行参，前两个用于将 c-mex 函数的输出传输到 MATLAB，后两个用于将 MATLAB 的输入传输到 c-mex 函数：

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
```

步骤 1　创建新文件，输入如下代码并保存为 conv2Mex.cpp：

```
#include "mex.h"

void conv2Mex(float* src, float* dst, int numRows, int numCols,
float* mask)
{
    int boundCol = numCols - 1;
    int boundRow = numRows - 1;

    for (int c = 1; c < boundCol; c++)
    {
        for (int r = 1; r < boundRow - 1; r++)
        {
            int dstIndex = c * numRows + r;
            int kerIndex = 8;
            for (int kc = -1; kc < 2; kc++)
            {
                int srcIndex = (c + kc) * numRows + r;
                for (int kr = -1; kr < 2; kr++)
                    dst[dstIndex] += mask[kerIndex--] * src
                    [srcIndex + kr];
            }
        }
    }
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, mxArray
*prhs[])
{
    if (nrhs != 2)
        mexErrMsgTxt("Invaid number of input arguments");

    if (nlhs != 1)
        mexErrMsgTxt("Invalid number of outputs");

    if (!mxIsSingle(prhs[0]) && !mxIsSingle(prhs[1]))
        mexErrMsgTxt("input image and mask type must be single");
```

```
34
35     float* image = (float*)mxGetData(prhs[0]);
36     float* mask = (float*)mxGetData(prhs[1]);
37
38     int numRows = (int)mxGetM(prhs[0]);
39     int numCols = (int)mxGetN(prhs[0]);
40     int numKRows = (int)mxGetM(prhs[1]);
41     int numKCols = (int)mxGetN(prhs[1]);
42
43     if (numKRows != 3 || numKCols != 3)
44         mexErrMsgTxt("Invalid mask size. It must be 3×3");
45
46     plhs[0] = mxCreateNumericMatrix(numRows, numCols,
       mxSINGLE_CLASS, mxREAL);
47     float* out = (float*)mxGetData(plhs[0]);
48
49     conv2Mex(image, out, numRows, numCols, mask);
50 }
```

在 mexFunction 中，先检查输入和输出的个数。本例中，必须有两个输入（图像和掩膜）和一个输出（卷积结果）。然后确保输入数据类型为 single。再确定输入图像和掩膜的大小，确保掩膜大小为 3×3。在第 46 行，准备存储运行结果的输出数组，并将其传回 MATLAB。然后调用我们编写的卷积函数 conv2Mex 进行数字运算。

步骤 2 编译我们编写的 mex 函数，并从 MATLAB 命令窗口调用该函数。编译十分简单：

```
>> mex conv2Mex.cpp
```

如果编译成功，mex 会生成一个二进制文件，在 64 位 Windows 操作系统中文件名为 conv2Mex.mexw64。

步骤 3 我们能够在 MATLAB 中任意地方调用这个函数，并得到结果：

```
>> quaters = (single)imread('eight.tif');
>> mask = single([1 2 1; 0 0 0; -1 -2 -1]);
>> H2 = conv2Mex(quarters, mask);
>> imagesc(H2);
>> colormap(gray);
```

可以使用 convol_mex.m 运行上述整个过程，代码如下：

```
% convol_mex.m

mex conv2Mex.cpp

quarters = single(imread('eight.tif'));
mask = single([1 2 1; 0 0 0; -1 -2 -1]);
imagesc(quarters);
colormap(gray);
```

```
H2 = conv2Mex(quarters, mask);
imagesc(H2);
colormap(gray);
```

2.6.3　在编写的 c-mex 中利用 CUDA 计算卷积

本例将使用 CUDA 函数进行卷积运算。除了通过 CUDA 实现以外，该 CUDA 函数和上节的 c-mex 函数功能相同。

步骤 1　定义在 CUDA 函数中调用函数的原型。创建新文件并保存为 conv2Mex.h：

```
#ifndef __CONV2MEXCUDA_H__
#define __CONV2MEXCUDA_H__

extern void conv2Mex(float* in,
                     float* out,
                     int numRows,
                     int numCols,
                     float* mask);

#endif // __CONV2MEXCUDA_H__
```

步骤 2　在 CUDA 中执行 conv2Mex 函数。创建 conv2Mex.cu 并输入如下代码：

```
#include "conv2Mex.h"

__global__ void conv2MexCuda(float* src,
                             float* dst,
                             int numRows,
                             int numCols,
                             float* mask)
{
    int row = blockIdx.x;
    if (row < 1 || row > numRows - 1)
        return;

    int col = blockIdx.y;
    if (col < 1 || col > numCols - 1)
        return;

    int dstIndex = col * numRows + row;
    dst[dstIndex] = 0;
    int kerIndex = 3 * 3 - 1;
    for (int kc = -1; kc < 2; kc++)
    {
        int srcIndex = (col + kc) * numRows + row;
        for (int kr = -1; kr < 2; kr++)
        {
            dst[dstIndex] += mask[kerIndex--] * src
            [srcIndex + kr];
```

```
            }
        }
    }

    void conv2Mex(float* src, float* dst, int numRows, int numCols,
    float* ker)
    {
        int totalPixels = numRows * numCols;
        float *deviceSrc, *deviceKer, *deviceDst;

        cudaMalloc(&deviceSrc, sizeof(float) * totalPixels);
        cudaMalloc(&deviceDst, sizeof(float) * totalPixels);
        cudaMalloc(&deviceKer, sizeof(float) * 3 * 3);

        cudaMemcpy(deviceSrc,
                   src,
                   sizeof(float) * totalPixels,
                   cudaMemcpyHostToDevice);

        cudaMemcpy(deviceKer,
                   ker,
                   sizeof(float) * 3 * 3,
                   cudaMemcpyHostToDevice);

        cudaMemset(deviceDst, 0, sizeof(float) * totalPixels);

        dim3 gridSize(numRows, numCols);

        conv2MexCuda<<<gridSize, 1>>>(deviceSrc,
                                      deviceDst,
                                      numRows,
                                      numCols,
                                      deviceKer);

        cudaMemcpy(dst,
                   deviceDst,
                   sizeof(float) * totalPixels,
                   cudaMemcpyDeviceToHost);

        cudaFree(deviceSrc);
        cudaFree(deviceDst);
        cudaFree(deviceKer);
    }
```

使用 cudaMalloc 函数分配 CUDA 设备的内存。函数 CudaMemcpy 根据第四个参数从主机复制数据到设备，或者从设备复制数据到主机。然后在 CUDA 设备上，给输入图像、输出图像和掩膜分配内存。使用 cudaMemset，初始化输出数据为零。使用 conv2MexCuda 进行 CUDA 调用。这里简单地设置网格大小和图像大小相同。在 conv2MexCuda 中，每个 CUDA 网格应用 3×3 掩膜，计算每个输出像素的卷积计算值。第 4 章将详解介绍网格大小的内容。

步骤 3　编译 CUDA 代码生成目标文件，随后将其链接到 c-mex 函数。在 MATLAB 命令窗口输入如下指令完成编译：

```
>> system('nvcc -c conv2Mex.cu')
```

如果遇到编译错误，请参考 2.5 节“实例：使用 CUDA 实现简单的向量加法”。编译成功后将会生成 conv2Mex.obj 文件。

步骤 4　接着创建 mex 函数，调用基于 CUDA 的卷积函数。创建一个新文件，输入如下代码，并保存为 conv2MexCuda.cpp：

```
1   #include "mex.h"
2   #include "conv2Mex.h"
3
4   void mexFunction(int nlhs, mxArray *plhs[], int nrhs, mxArray
    *prhs[])
5   {
6       if (nrhs != 2)
7           mexErrMsgTxt("Invaid number of input arguments");
8
9       if (nlhs != 1)
10          mexErrMsgTxt("Invalid number of outputs");
11
12      if (!mxIsSingle(prhs[0]) && !mxIsSingle(prhs[1]))
13          mexErrMsgTxt("input image and mask type must be single");
14
15      float* image = (float*)mxGetData(prhs[0]);
16      float* mask = (float*)mxGetData(prhs[1]);
17
18      int numRows = (int)mxGetM(prhs[0]);
19      int numCols = (int)mxGetN(prhs[0]);
20      int numKRows = (int)mxGetM(prhs[1]);
21      int numKCols = (int)mxGetN(prhs[1]);
22
23      if (numKRows != 3 || numKCols != 3)
24          mexErrMsgTxt("Invalid mask size. It must be 3×3");
25
26      plhs[0] = mxCreateNumericMatrix(numRows, numCols,
        mxSINGLE_CLASS, mxREAL);
27      float* out = (float*)mxGetData(plhs[0]);
28
29      conv2Mex(image, out, numRows, numCols, mask);
30  }
```

新的 mex 函数和前一章的基本相同。唯一区别在于第一行#include "conv2Mex.h"。这里，我们在 conv2Mex.h 中定义 conv2Mex 函数，在 conv2Mex.cu 中执行该函数。

步骤 5　这里基于 CUDA 的 mex 函数已经准备好。由于 conv2Mex 函数在 conv2Mex.obj 中，所以必须告知链接器函数的位置。同时，也告知链接器将要使

用 CUDA 运行时库及其位置。在 MATLAB 命令窗口输入如下命令。

Windows 64 位操作系统

```
>> mex conv2MexCuda.cpp conv2Mex.obj -lcudart -L"C:\Program Files
\NVIDIA GPU Computing Toolkit\CUDA\v5.0\lib\x64"
```

Windows 32 位操作系统

```
>> mex conv2MexCuda.cpp conv2Mex.obj -lcudart -L"C:\Program Files
\NVIDIA GPU Computing Toolkit\CUDA\v5.0\lib\Win32"
```

Linux 操作系统

```
>> mex conv2MexCuda.cpp conv2Mex.obj -lcudart -L"/usr/local/cuda/lib"
```

Mac OS X 操作系统

```
>> mex conv2MexCuda.cpp conv2Mex.obj -lcudart -L"/Developer/
NVIDIA/CUDA-5.0/lib"
```

成功后，MATLAB 会生成 mex 函数，在 Windows 64 位操作系统中为 convMexCuda.mexw64。

步骤 6 在 MATLAB 命令窗口执行基于 CUDA 的卷积函数：

```
>> quarters = single(imread('eight.tif'));
>> mask = single([1 2 1; 0 0 0; -1 -2 -1]);
>> H3 = conv2MexCuda(quarters, mask);
>> imagesc(H3);
>> colormap(gray);
```

此时，你会在 MATLAB 中看见相同的输出图像。

可以使用 convol_cuda.m 运行上述整个过程，代码如下：

```
% convol_cuda.m

system('nvcc -c conv2Mex.cu -ccbin "C:\Program Files (x86)\Microsoft
Visual Studio 10.0\VC\bin')
mex conv2MexCuda.cpp conv2Mex.obj -lcudart -L"C:\Program Files\NVIDIA
GPU Computing Toolkit\CUDA\v5.0\lib\x64"

quarters = single(imread('eight.tif'));
mask = single([1 2 1; 0 0 0; -1 -2 -1]);
imagesc(quarters);
colormap(gray);

H3 = conv2MexCuda(quarters, mask);
imagesc(H3);
colormap(gray);
```

2.6.4　简单的时间性能分析

通过上述 3 种方法完成二维卷积运算，可以得到相同的输出图像。我们主要关心三种方法在耗时方面的性能。更多详细的指令和信息请参见第 3 章。这里，在 MATLAB 中使用 tic 和 toc 指令进行简单的时间性能分析。

下面是采用不同方法时，示例 MATLAB 会话的时间性能：

```
>> tic;  H1 = conv2(quaters, mask); toc;
Elapsed time is 0.001292 seconds.
>> tic;  H1 = conv2(quaters, mask); toc;
Elapsed time is 0.001225 seconds.
>> tic;  H2 = conv2Mex(quaters, mask); toc;
Elapsed time is 0.001244 seconds.
>> tic;  H2 = conv2Mex(quaters, mask); toc;
Elapsed time is 0.001118 seconds.
>> tic;  H3 = conv2MexCuda(quaters, mask); toc;
Elapsed time is 0.036286 seconds.
>> tic;  H3 = conv2MexCuda(quaters, mask); toc;
Elapsed time is 0.035877 seconds.
```

内置的 conv2 函数和我们编写的 conv2Mex 函数的性能相同。然而，本例中基于 CUDA 的卷积要比上述两个函数慢得多。这是因为处理的数据尺寸很小，而且网格和线程块的尺寸未能利用 GPU 架构的优势。

	conv2	conv2Mex	Conv2MexCuda
耗时/s	0.00123	0.00122	0.0358

后续章节将会介绍如何获得更详细的时间分析结果，以及如何优化简单的 CUDA 函数来实现加速。

2.7　总结

与在 AddVectors 中所做的相同，将基于 CUDA 的函数置于 conv2Mex.cu 中（见图 2.14）。文件 conv2Mex.h 包含 mex 函数定义，conv2MexCuda.cpp 调用子例行程序。编译 CUDA 代码为目标文件后，使用 mex 指令来编译 mex 代码并链接到 CUDA 目标文件。

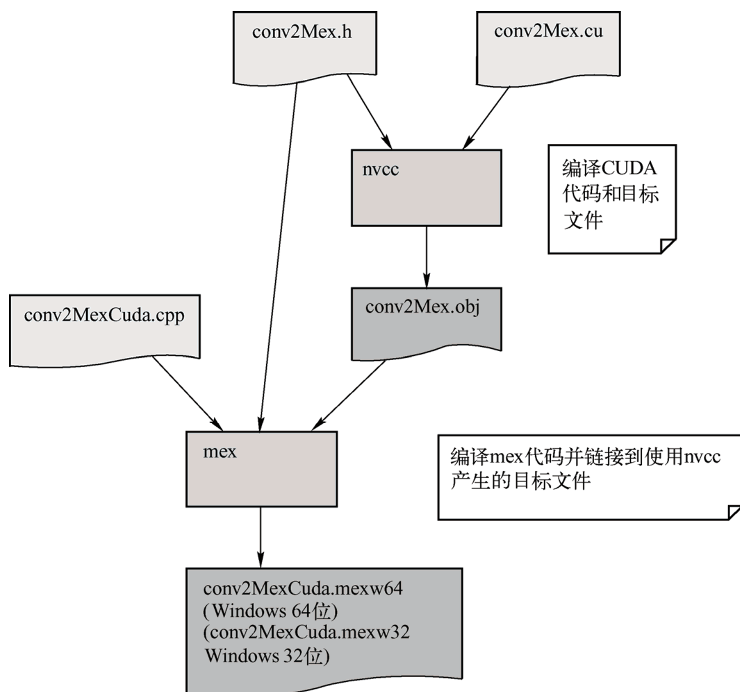


图 2.14 示例的总结框图

（□：输入源文件，□：指令，□：生成文件）

第 3 章　通过耗时分析进行最优规划

3.1　本章学习目标

通过分析，可以找出代码中较为耗时的部分，确定代码瓶颈。由于较大的程序实际中往往包括多层，不能直接依次查找函数，而需要调用其他一些耗时的函数。而且可能会在代码较低层遇到这种情况。通过分析的过程，可以有效判断哪些代码是负责上述情况调用的。对于优化规划来说，这是关键的一步。在本章，你可以了解以下内容：

- 使用 MATLAB 内置分析器查找 m 文件的瓶颈。
- 采用 C/C++分析方法进行 c-mex 代码分析。
- 使用 Visual Studio 和 NVIDIA Visual Profiler 进行 CUDA 代码分析。
- c-mex 调试器的环境设置。

3.2　分析 MATLAB 代码查找瓶颈

3.2.1　分析器的使用方法

幸运的是，MATLAB 提供了相当易于使用的分析器（profiler）。下面再次使用第 2 章的二维卷积范例进行时间分析范例。

你可以通过两种方式调用 MATLAB 分析器。第一种方法是在 MATLAB Desktop 中选择 Profiler（见图 3.1）。第二种方法是在命令窗口输入“profile viewer”：

```
>> profile viewer
```

这样，可以得到如图 3.2 所示的分析器窗口。

为了使用二维卷积范例，在使用分析器之前，要在 MATLAB 主窗口下更改当前文件夹，如图 3.3 所示。然后当需要运行分析器时，在“Run This Code：”处输入想要运行的程序。这里以 convQuarterImage.m 为例，如图 3.4 所示。

可以得到如图 3.5 所示的分析结果。在每一列索引中，可以点击 Function Name、Calls、Total Time 和 Self Time，根据索引对结果进行排序。

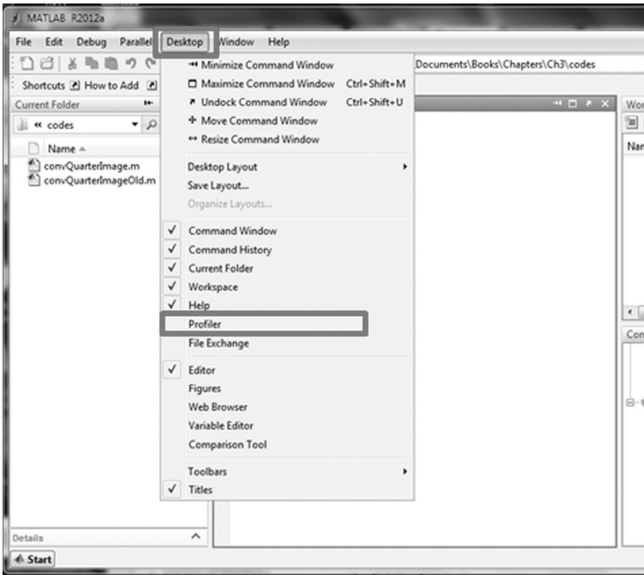


图 3.1　从 MATLAB desktop 菜单选择分析器

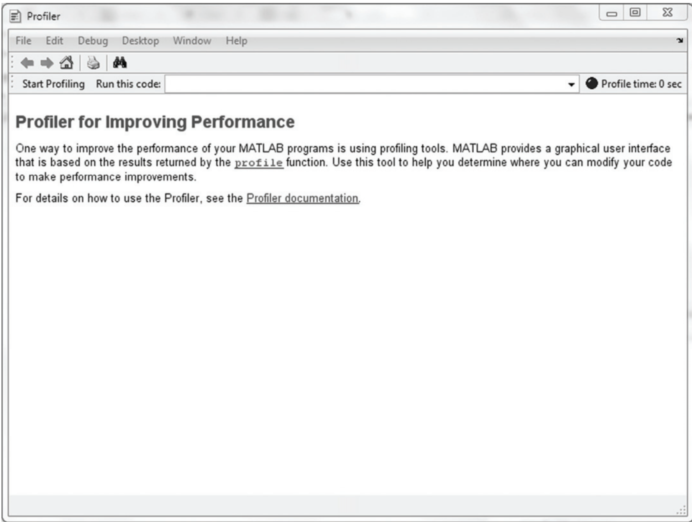


图 3.2　MATLAB 分析器窗口

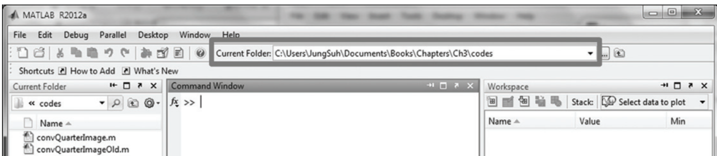


图 3.3　在 MATLAB 主窗口下更改当前文件夹

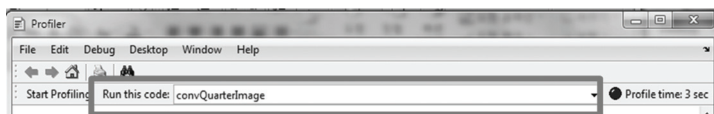


图 3.4　MATALB 分析器中 Run this code 栏

如果在 Function Name 列中点击 convQuarterImage，就可以得到 convQuarter Image 文件中每一行更详细的耗时信息，如图 3.6 所示。

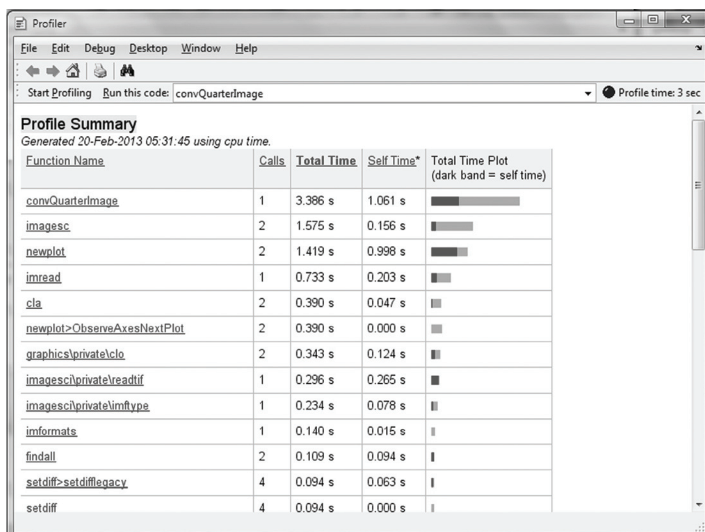


图 3.5　MATLAB 分析器中的耗时分析总结

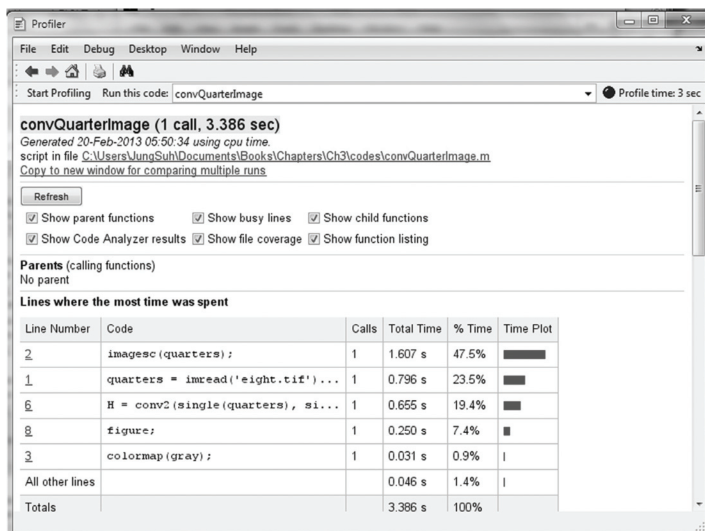


图 3.6　分析结果中的更多细节

向下滚动这个窗口时，可以看到根据各个类别（时间、总调用次数、覆盖等）高亮标识的代码。根据分析结果（见图 3.7）可以看到 imagesc( ) 和 imread( ) 占用 convQuarterImage 大部分的运行时间。因为 imread( ) 是读取输入图像的函数，imagesc( ) 是缩放或者显示输入图像的函数，所以可以只关注下面的纯计算部分，这部分耗时更多：

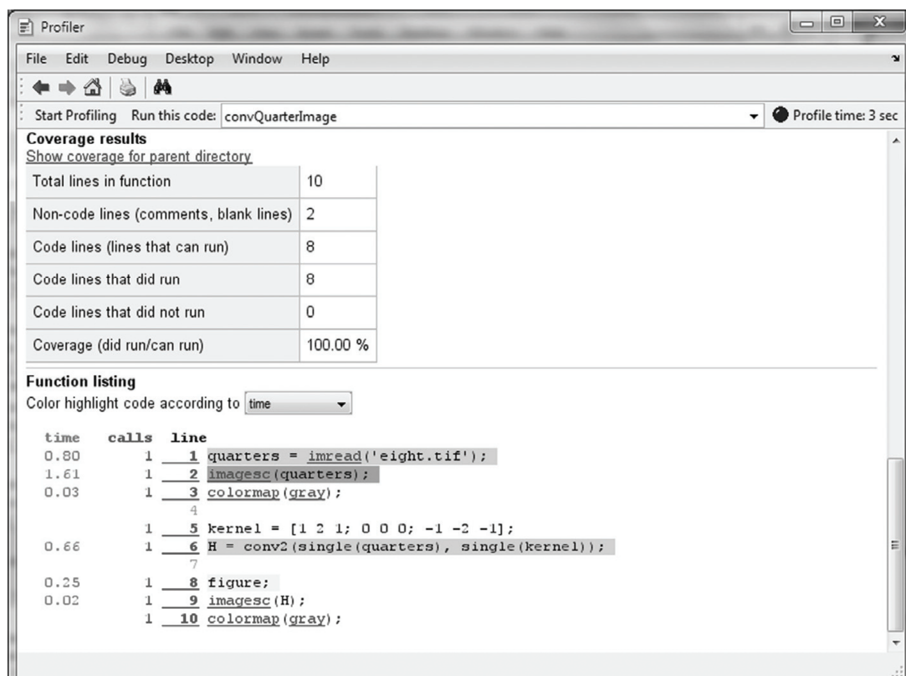


图 3.7　分析结果中每一行的耗时信息

```
H = conv2(single(quarters), single(kernel));
```

3.3 节可以看到用 c-mex 替换这一行时的分析结果，并了解 c-mex 中的 C/C++ 分析方法。

3.2.2　针对多核 CPU 更精确的耗时分析

如今，多核 CPU 计算机十分普遍，程序运行速度与可用的内核数量及它们的可用性直接相关。如果只想要粗略地了解函数调用次数和耗时，而不需要了解 CPU 占用率更精确的信息，那么使用前面介绍的分析器设置方法就可以满足要求了。然而在某些情况下，需要更精确地分析程序的使用情况。要做到这一点，就需要在 MATLAB 之外，手动地设置可以使用的内核数量，分析需要精确测量的代码。在 Windows 操作系统中，很容易控制 CPU 内核使用数量，如图 3.8 所示。

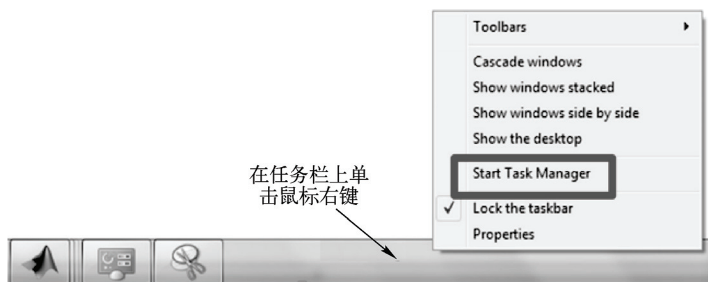


图 3.8　控制 CPU 内核使用数量，打开“启动任务管理器”（Start Task Manager）

在任务栏上单击鼠标右键，可以打开“启动任务管理器”（Start Task Manager）菜单。单击启动任务管理器，打开“Windows 任务管理器”（Windows Task Manager）窗口，如图 3.9 所示。

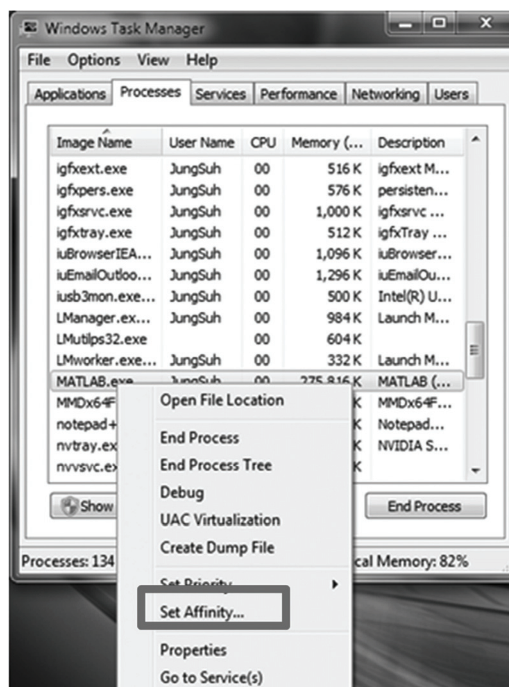


图 3.9　控制 CPU 内核使用数，单击“设置相关性”（Set Affinity...）

对 MATLAB 进程，单击右键，选择“设置相关性”（Set Affinity...）。然后进入各处理器的选择访问窗口（见图 3.10），你可以选择使用某个特定的处理器来分析代码。选择一个处理器运行 MATLAB.exe，而关闭其他处理器，可以更精确地分析代码，如图 3.11 所示。

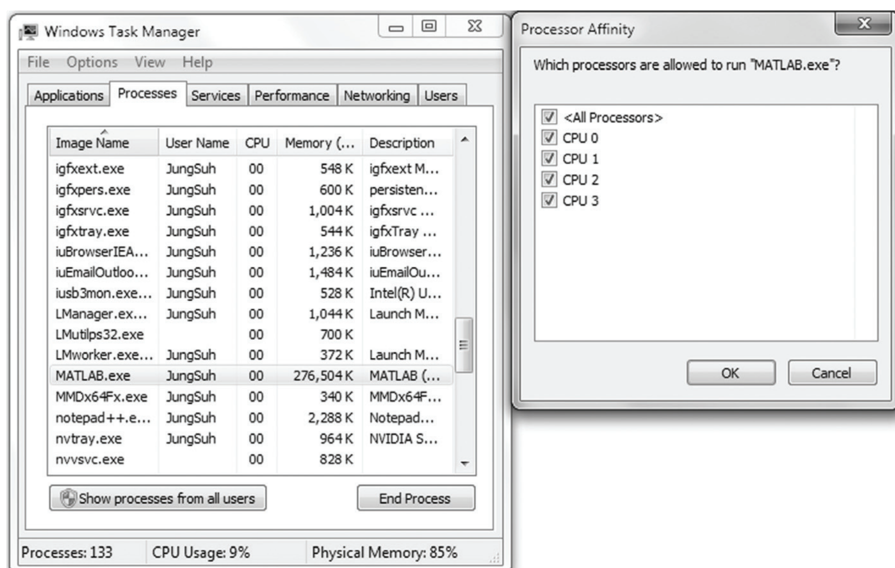


图 3.10 控制 CPU 内核使用数，单击“处理器相关性”（Processor Affinity）窗口

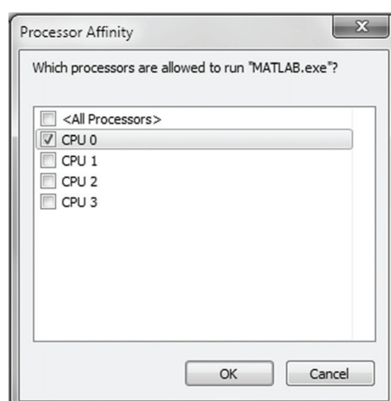


图 3.11 控制 CPU 内核使用数量，选择要使用的处理器

3.3 CUDA 的 c-mex 代码分析

3.3.1 利用 Visual Studio 进行 CUDA 分析

对于安装了 Microsoft Visual Studio 的 CUDA，NVIDIA Nsight（Visual Studio 版）是一个免费的开发环境。NVIDIA Nsight（Visual Studio 版）提供了强大的调试和分析函数，这对 CUDA 代码开发非常有效。NVIDIA Nsight 的下载和安装请

参考附录 B。

下面利用 CUDA 重温卷积的范例。在第 2 章中，使用 CUDA 函数创建了一个卷积函数，在 MATLAB 命令窗口中运行，如下所示：

```
>> quarters = single(imread('eight.tif'));
>> mask = single([1  2  1;  0  0  0;  -1  -2  -1]);
>> H3 = conv2MexCuda(quarters, mask);
>> imagesc(H3);
>> colormap(gray);
```

使用 3.2 节介绍的方法，打开 Visual Studio，如图 3.12 所示。

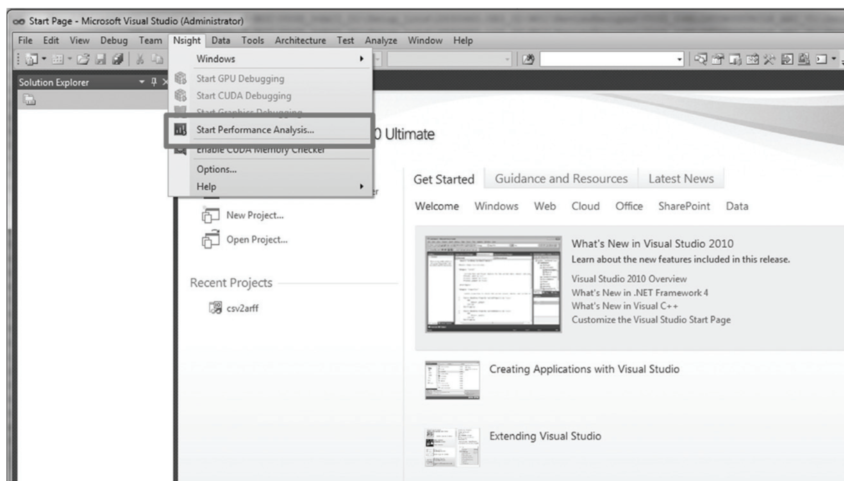


图 3.12　Microsoft Visual Studio 中安装的 Nsight

单击 Nsight 菜单，选择 Start Performance Analysis...。弹出对话框，询问是否在不安全状态下连接（见图 3.13）。选择 Connect unsecurely，单击 OK 按钮，就会在屏幕上显示 Visual Studio，如图 3.14 所示。

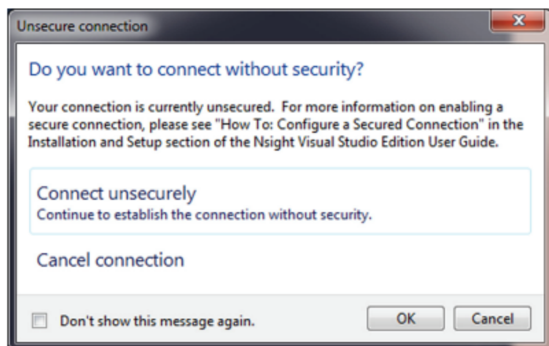


图 3.13　用于连接 Nsight 和 MATLAB 的 Unsecure connection 对话框

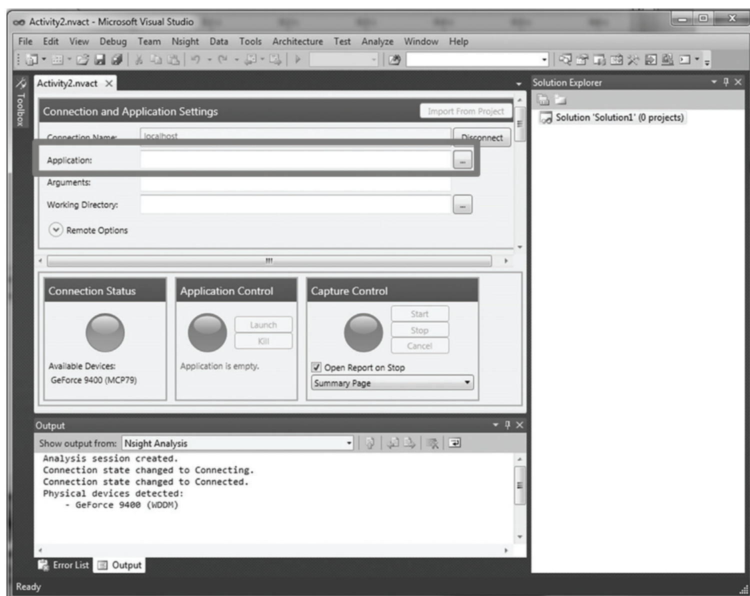


图 3.14　Visual Studio 中应用程序链接窗口

在 Application 处，单击文件夹浏览按钮，选择 MATLAB 可执行文件。MATLAB 可执行文件可以在 MATLAB 安装目录下找到。需要根据系统架构，选择一个适合的可执行文件。例如，在 Windows 7 系统中，64 位 MATLAB 可以在图 3.15 所示位置找到可执行文件。

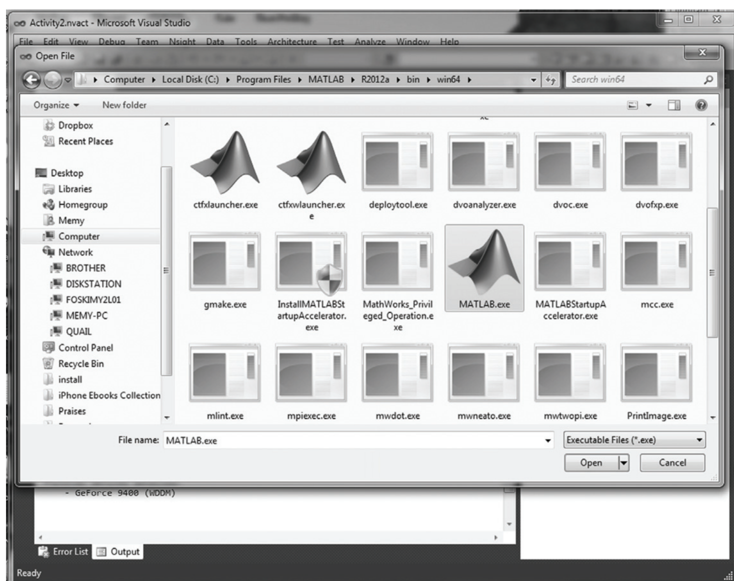


图 3.15　选择 MATLAB 作为连接应用

选择 MATLAB.exe，然后单击 Open 按钮，关闭对话框。现在，向下滚动一点，到 Activity Type 处，单击选中 Profile CUDA Application 选项，如图 3.16 所示。选择此选项后，可以看到 Application Control 中 Launch 按钮可用。

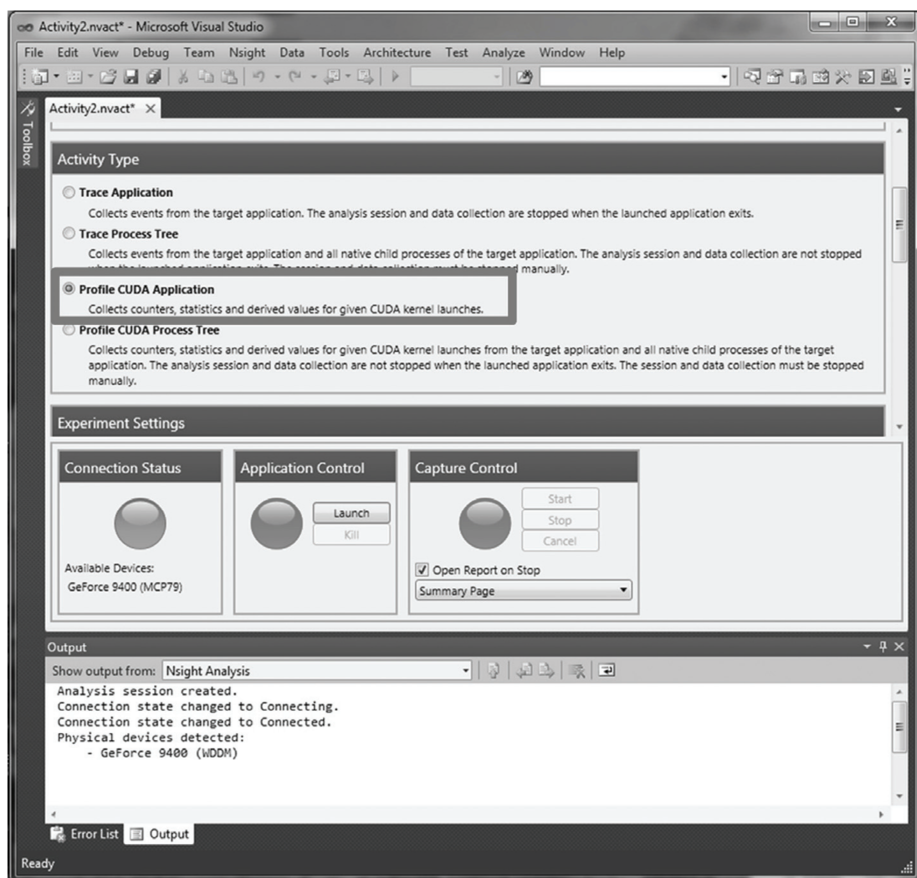


图 3.16　选择“Profile CUDA Application”作为作业类型

单击 Launch 按钮，然后可以看到 MATLAB 开始和 Visual Studio 一起运行，如图 3.17 所示。

在 MATLAB 命令窗口中，运行基于 CUDA 的卷积程序（见图 3.18），然后回到 Visual Studio 中，在 Capture Control（捕获控制）对话框中单击 Stop 按钮（见图 3.19）。停止捕获后，可以看到 CUDA Overview（见图 3.20）。在 CUDA 标题栏中单击链接 Launches，此时会显示出 CUDA 函数和所有内核函数细节和时间特性（见图 3.21）。如果选择 conv2MexCuda [CUDA Kernel]，你可以看到所设置的网格和线程块的尺寸，以及完成这个任务用了多少时间（见图 3.22）。

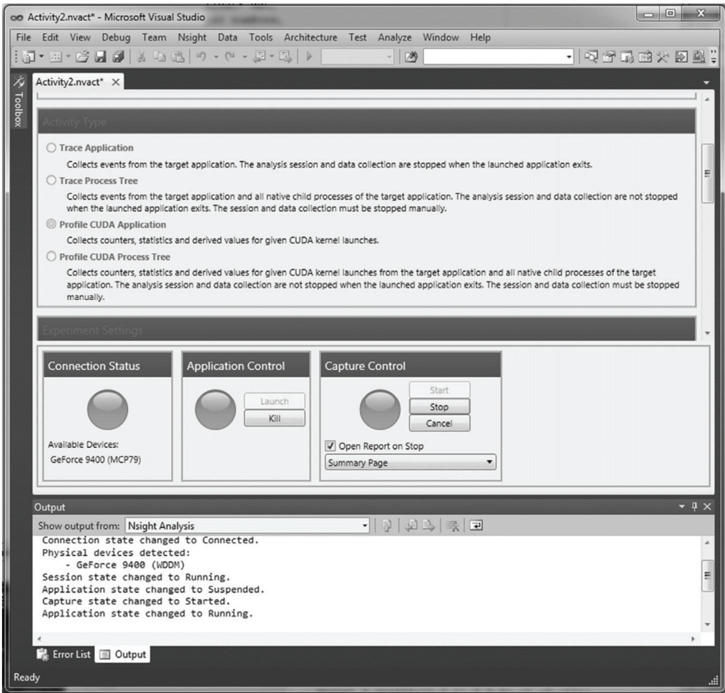


图 3.17 选择 Launch 之后的应用程序

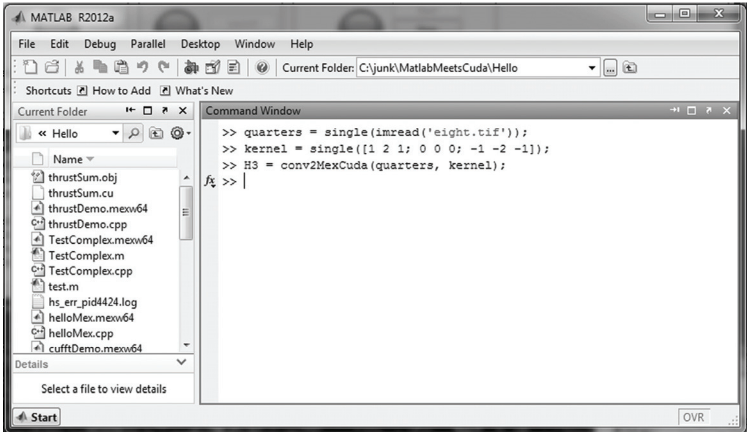


图 3.18 运行 MATLAB 作为分析应用程序

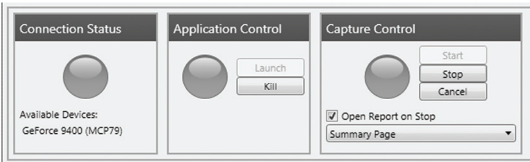


图 3.19 在 Visual Studio 内完成 MATLAB 分析

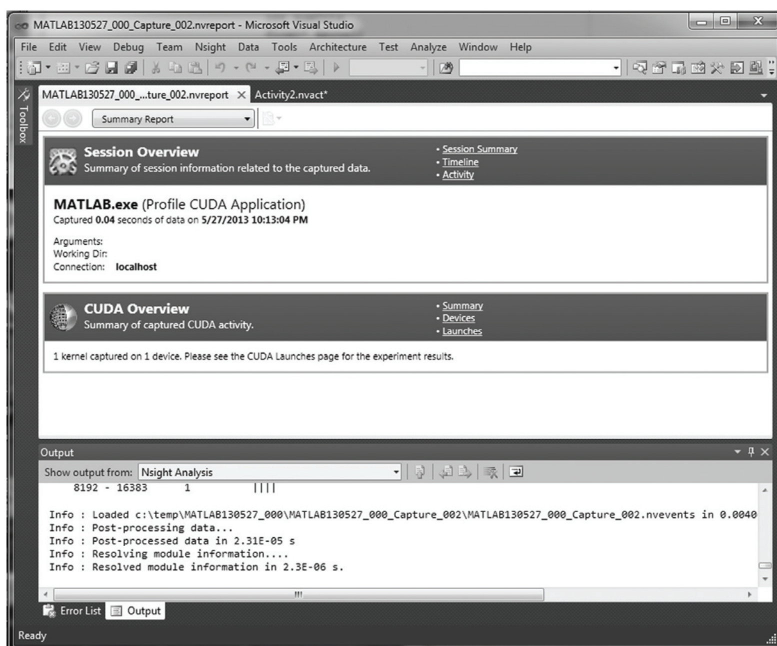


图 3.20　完成分析后的 CUDA Overview

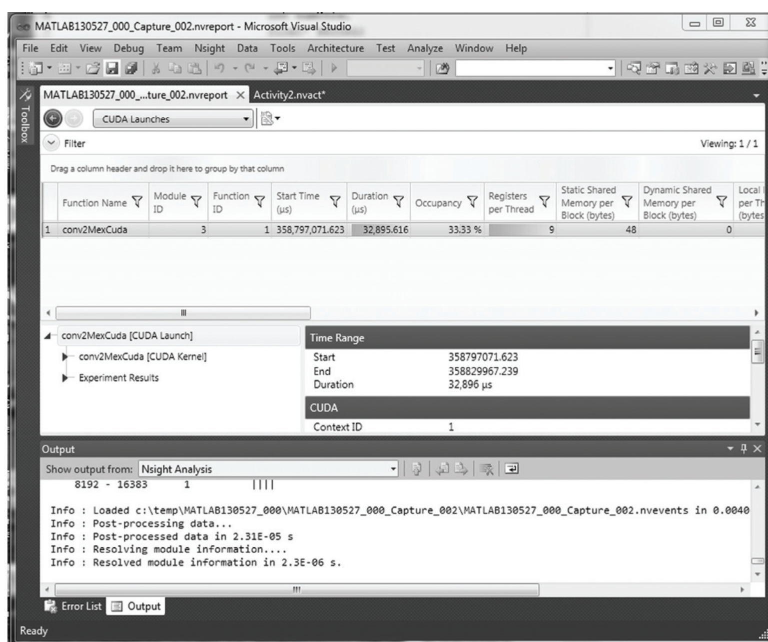


图 3.21　Nsight 窗口中 CUDA 内核函数细节和时间分析结果

图 3.22　Nsight 窗口中 CUDA 内核函数和时间分析更多的细节信息

返回到 Activity 选项卡，并在 Capture Control 对话框中单击 Start 按钮，可以重复这个分析过程（见图 3.23）。如果这么做，你需要关闭 MATLAB 窗口或者在 Application Control 对话框中单击 Kill 按钮。这样可以关闭在 Visual Studio 中进行分析的整个会话。

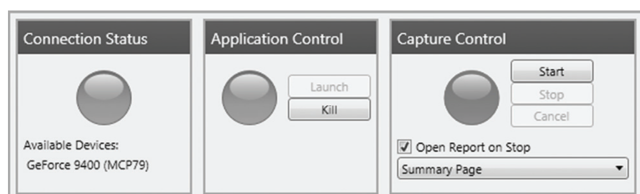


图 3.23　Nsight 中 Capture Control

3.3.2　利用 NVIDIA Visual Profiler 进行 CUDA 分析

NVIDIA Visual Profiler 提供了丰富的图形用户环境，可以给出 CUDA 在后台工作的更多细节。除了提供每个 CUDA 函数调用的时间分析外，它还能给出如何调用内核函数以及存储器的使用情况等。它有助于定位瓶颈可能出现的位置，并详细解释如何调用内核。

本节将展示这一出色的工具如何与 MATLAB 和 CUDA 一起使用。NVIDIA Visual Profiler 可以在 CUDA 安装目录下找到（见图 3.24）。对于 Windows 操作系统，通常在 C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\libnvvp 目录下。对于 Mac OS X 操作系统，NVIDIA Visual Profiler 的位置如图 3.25 所示。对于 Linux 发行版，位于/usr/local/cuda/libnvvp 目录下（见图 3.26）。启动 nvvp，会先得到一个空窗口。

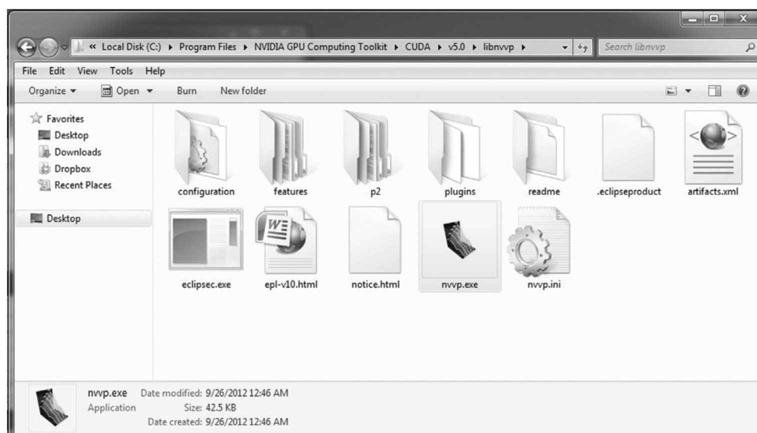


图 3.24　NVIDIA Visual Profiler 和 CUDA 安装位置

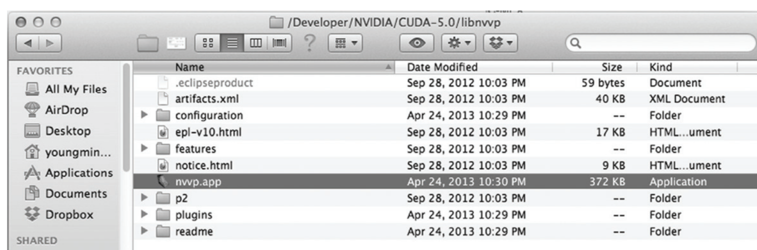


图 3.25　Mac OS X 操作系统中的 NVIDIA Visual Profiler

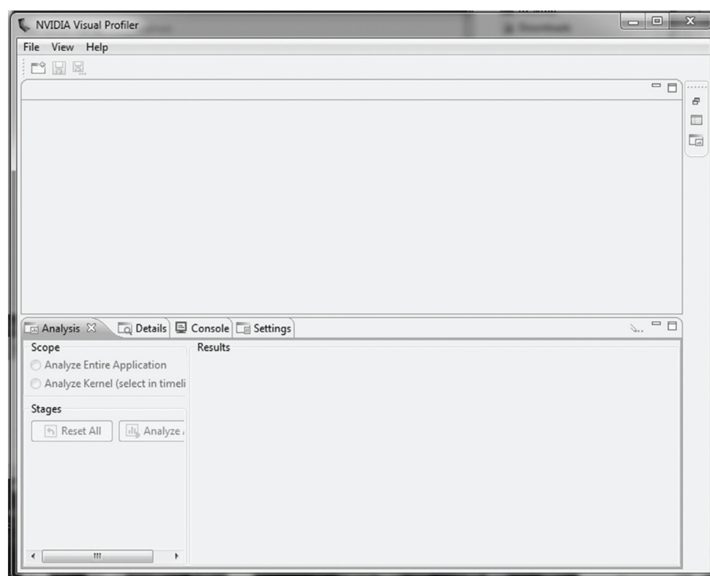


图 3.26　Linux 操作系统中的 NVIDIA Visual Profiler

首先，打开 NVIDIA Visual Profiler。然后，在主菜单中单击 File >New Session 菜单命令，创建一个新的会话，如图 3.27 所示。

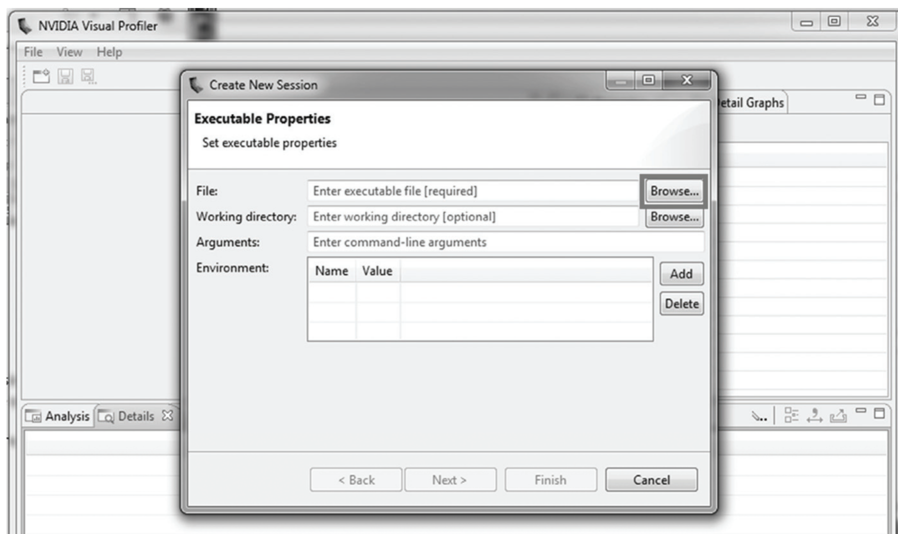


图 3.27　NVIDIA Visual Profiler 中的 New Session

单击 Browse 按钮，与之前一样，选择 MATLAB 可执行文件，如图 3.28 所示。MATLAB 可执行文件是在 MATLAB 的 bin 安装目录下，实际位置取决于你的系统架构：

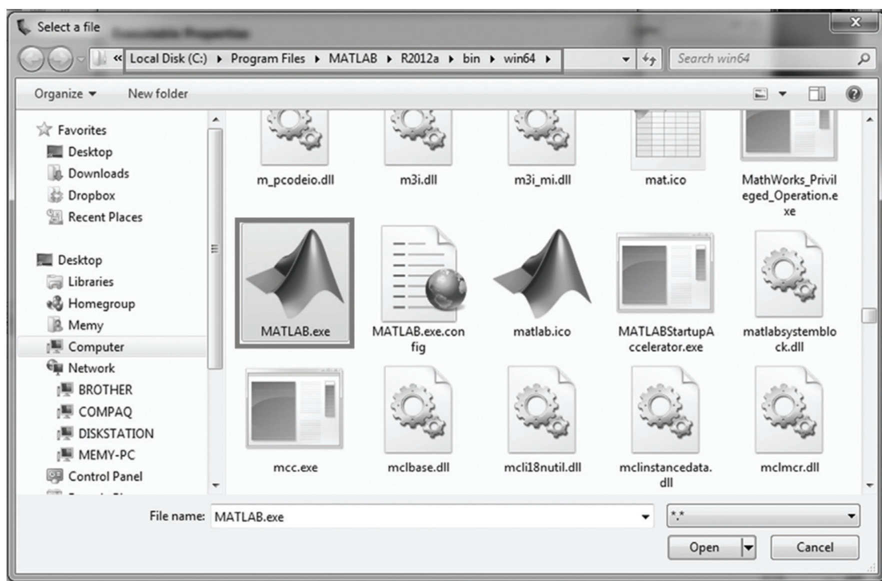


图 3.28　为 NVIDIA Visual Profiler 选择 MATLAB 可执行文件

● 对于 Windows 64 位操作系统

C:\Program Files\MATLAB\R2012a\bin\x64\MATLAB.exe.

● 对于 Windows 32 位操作系统

C:\Program Files\MATLAB\R2012a\bin\win32\MATLAB.exe.

● 对于 Mac OS X 操作系统

/Applications/MATLAB_R2012a.app/bin/maci64/MATLAB.

● 对于 Linux 操作系统

/usr/local/MATLAB/R2012a/bin/glnxa64/MATLAB.

基于系统架构选择 MATLAB 可执行文件后，单击文件选择对话框中的 Open 按钮，回到 Create New Session 对话框（见图 3.29）。单击 Create New Session 对话框中的 Next 按钮，然后选择可执行文件的属性（见图 3.30）。现在，所有选择都设为默认值，单击 Finish 按钮完成创建新会话的过程。完成这些后，NVIDIA Visual Profiler 启动 MATLAB 可执行程序（见图 3.31）。然后，等待直至 MATLAB 关闭。

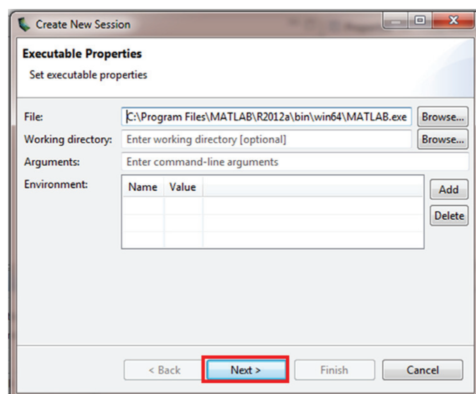


图 3.29　Create New Session 会话窗口

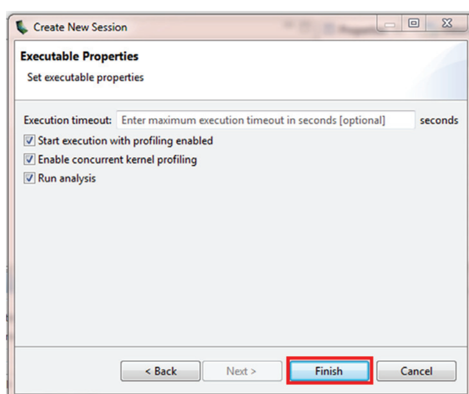


图 3.30　Executable Properties 选项

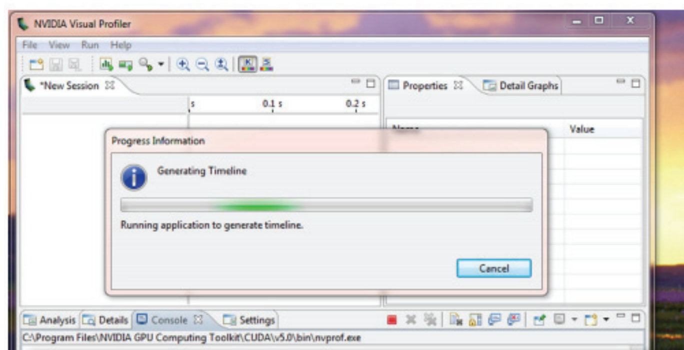


图 3.31　在 NVIDIA Visual Profiler 中启动 MATLAB 进行分析

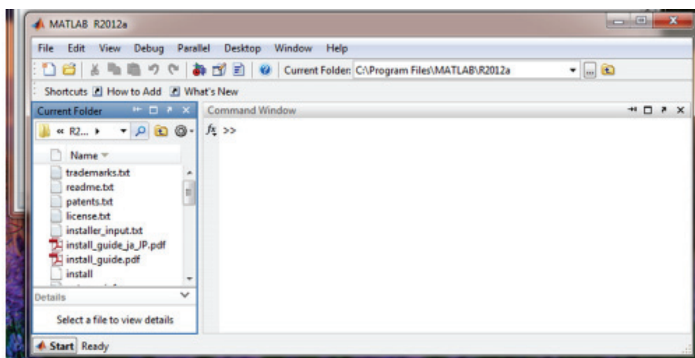


图 3.31 在 NVIDIA Visual Profiler 中启动 MATLAB 进行分析（续）

在 MATLAB 命令窗口中，运行基于 CUDA 的卷积程序如下：

```
>> quaters = (single)imread('eight.tif');
>> mask = single([1  2  1;  0  0  0;  -1  -2  -1]);
>> H3 = conv2MexCuda(quarters, mask);
```

运行这些指令之后，关闭 MATLAB 窗口，分析器将开始生成分析数据。此时如果遇到如图 3.32 所示的警告信息，可以通过在 c-mex 函数末尾添加 cudaDeviceReset( ) 语句稍微修改代码，以确保清除所有分析数据。

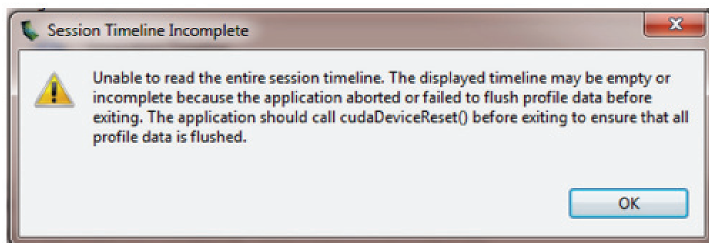


图 3.32 应用程序未完成警告信息

```
#include "mex.h"
#include "conv2Mex.h"
#include <cuda_runtime.h>
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, mxArray *prhs[])
{

  float* out = (float*)mxGetData(plhs[0]);
  conv2Mex(image, out, numRows, numCols, kernel);
  cudaDeviceReset();

}
```

并以一个附加选项重新编译 c-mex：

```
>> mex conv2MexCuda.cpp conv2Mex.obj -lcudart -L"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\lib\x64" -I"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\include"
```

重新运行卷积程序，并关闭 MATLAB 窗口，NVIDIA Visual Profiler 会显示所有信息，如图 3.33 所示。

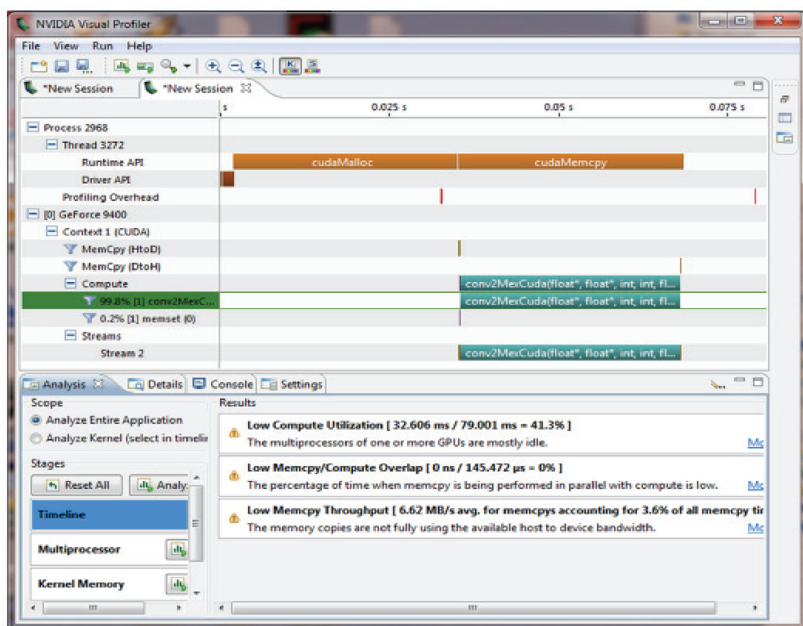


图 3.33　NVIDIA Visual Profiler 分析结果窗口

你可以对每个 CUDA 函数的耗时、GPU 占用情况等有很好的了解。单击底部的 Details 选项卡，可以看到网格和线程块中线程的数量，如图 3.34 所示。

Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput
Memcpy HtoD [sync]	35.143 ms	63.072 μs	n/a	n/a	n/a	n/a	n/a	2...	4.4 GB/s
Memcpy HtoD [sync]	35.345 ms	5.184 μs	n/a	n/a	n/a	n/a	n/a	3...	6.62 MB/s
memset (0)	35.357 ms	54.592 μs	n/a	n/a	n/a	n/a	n/a	2...	5.09 GB/s
conv2MexCuda(float*, float*, i...	35.415 ms	32.552 ms	[242,308,1]	[1,1,1]	9	48	0	n/a	n/a
Memcpy DtoH [sync]	67.968 ms	77.216 μs	n/a	n/a	n/a	n/a	n/a	2...	3.6 GB/s

图 3.34　NVIDIA Visual Profiler Details 选项卡下的时间信息

3.4　c-mex 调试器的环境设置

因为 MATLAB 只提供了 m 文件编译器和与 m 文件相关的工具，为了在 c-mex 文件中调试 C/C++代码，需要使用不同于 MATLAB 的调试器。用其他调试器来调试与 MATLAB 中 m 文件相关的 c-mex 文件也十分容易。在之前章节中，

我们使用 conv2d3×3.cpp 文件，这个由 convQuarterImageCmex.m 调用的 C++文件如下：

The `conv2d3×3.cpp` **File**

```
#include "mex.h"
#include "conv2d3x3.h"

void conv2d3x3(float* src, float* dst, int numRows, int numCols, float*
mask)
{
  int boundCol = numCols - 1;
  int boundRow = numRows - 1;

  for (int c = 1; c < boundCol; c++)
  {
    for (int r = 1; r < boundRow - 1; r++)
    {
      int dstIndex = c * numRows + r;
      int mskIndex = 8;
      for (int kc = -1; kc < 2; kc++)
      {
        int srcIndex = (c + kc) * numRows + r;
        for (int kr = -1; kr < 2; kr++)
          dst[dstIndex]+ = mask[mskIndex--] * src[srcIndex + kr];
      }
    }
  }
}
 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, mxArray *prhs[])
 {
   if (nrhs ! = 2)
      mexErrMsgTxt("Invaid number of input arguments");

   if (nlhs ! = 1)
      mexErrMsgTxt("Invalid number of outputs");


   if (!mxIsSingle(prhs[0]) && !mxIsSingle(prhs[1]))
      mexErrMsgTxt("input image and mask type must be single");

   float* image = (float*)mxGetData(prhs[0]);
   float* mask = (float*)mxGetData(prhs[1]);

   int numRows = mxGetM(prhs[0]);
   int numCols = mxGetN(prhs[0]);
   int numKRows = mxGetM(prhs[1]);
```

```
    int numKCols = mxGetN(prhs[1]);

    if (numKRows != 3 || numKCols != 3)
      mexErrMsgTxt("Invalid mask size. It must be 3x3");

    plhs[0] = mxCreateNumericMatrix(numRows, numCols, mxSINGLE_CLASS,
    mxREAL);
    float* out = (float*)mxGetData(plhs[0]);

    conv2d3x3(image, out, numRows, numCols, mask);
  }
```

The `convQuarterImageCmex.m` **File**

```
quarters = imread('eight.tif');
imagesc(quarters);
colormap(gray);

mask = [1 2 1; 0 0 0; -1 -2 -1];
single_q = single(quarters);
single_k = single(mask);

H = conv2d3x3(single_q, single_k);  % Call C-Mex file here

figure;
imagesc(H);
colormap(gray);
```

为了调试与 convQuarterImageCmex.m 文件相关的 conv2d3×3.cpp c-mex 文件，需要利用-g 选项在 MATLAB 命令窗口中编译 conv2d3×3.cpp 文件：

```
>> mex -g conv2d3×3.cpp
```

成功后，将在相同的目录下生成一个新的文件 conv2d3x3.mexw64（或者 conv2d3×3.mexw32）。然后，打开 Visual Studio，同时保持 MATLAB 会话，并在 Tools 菜单中选择 Attach to Process...（见图 3.35）。

在 Attach to Process 工具箱中，可以看到计算机上正在运行的可用进程（见图 3.36）。如果关闭 MATLAB，就无法在可用进程窗口中找到 MATLAB.exe。选择 MATLAB.exe 并单击 Attach 按钮。然后，Visual Studio 会出现一个顶部带有 Solution1 (Running) 的空窗口，如图 3.37 所示。在 Visual Studio 中，通过 File 菜单栏 Open 下的 File 选项打开 conv2d3×3.cpp c-mex 源文件（见图 3.38）。接下来，在你希望的一行单击右键，设置一个断点（见图 3.39）。然后，可以看到一个未激活的断点和警告消息，可以忽略该消息，如图 3.40 所示。一旦正确设置了断点，你就可以使用 Debug 菜单下的所有功能，而没有其他限制，如图 3.41 所示。

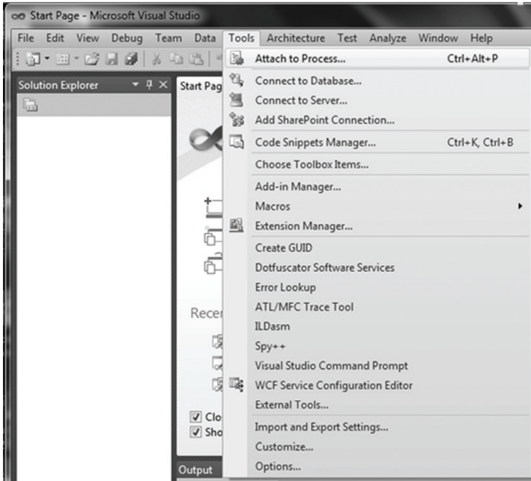


图 3.35 Microsoft Visual Studio 调试器中 Attach to Process...菜单

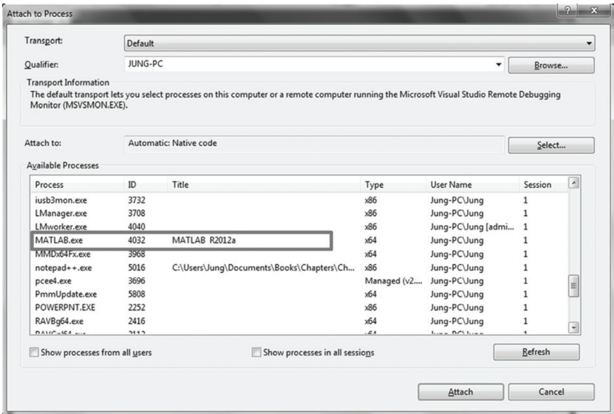


图 3.36 将 MATLAB 添加到 Microsoft Visual Studio 调试器中

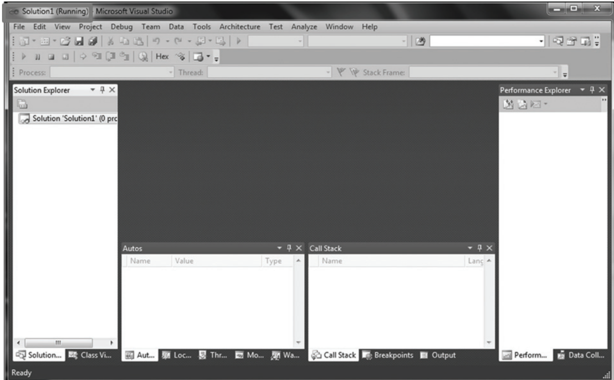


图 3.37 Microsoft Visual Studio 调试器中带有 Solution1 (Running)的空窗口

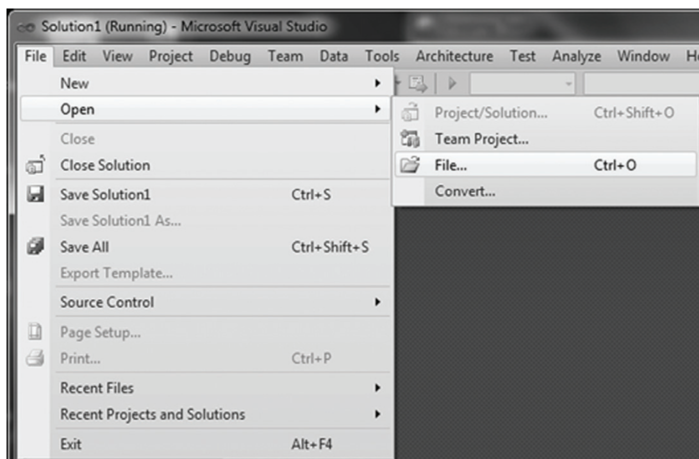


图 3.38　在 Microsoft Visual Studio 调试器中打开源代码

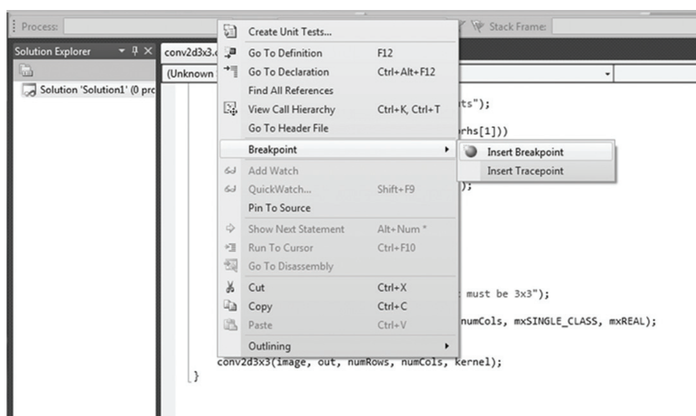


图 3.39　在调试器上插入一个断点

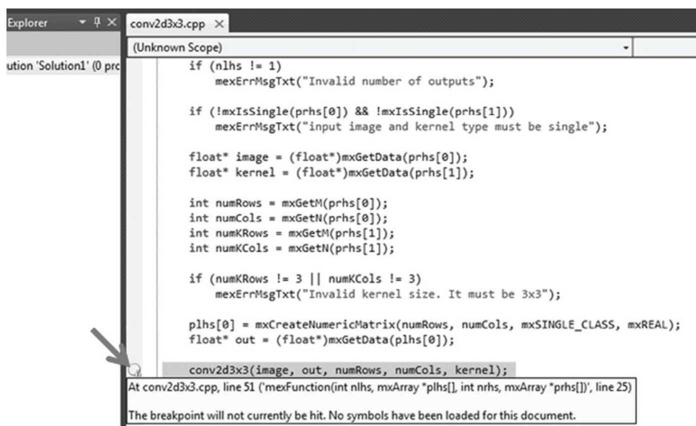


图 3.40　激活的断点

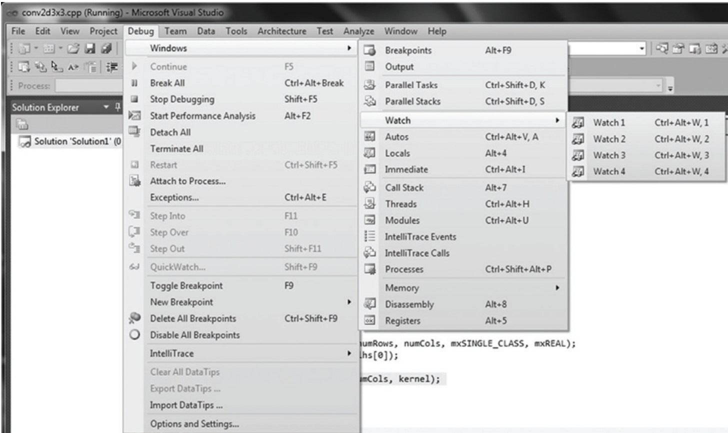


图 3.41 Microsoft Visual Studio 调试器的各种功能

现在，回到 MATLAB。在 MATLAB 命令窗口中运行调用 conv2d3×3.cpp c-mex 文件的 convQuarterImageCmex.m 文件，如图 3.42 所示。然后，Visual Studio 的调试模式自动激活，如图 3.43 所示，程序会运行到设置的断点处暂停。

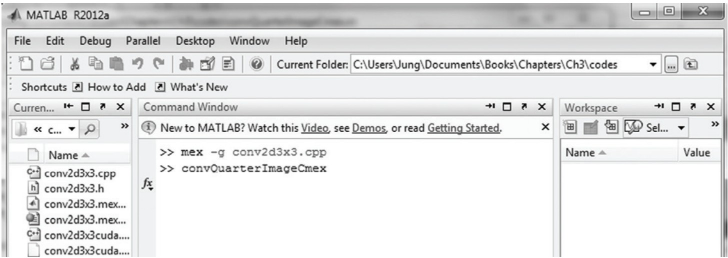


图 3.42 运行 MATLAB 主模块进行调试

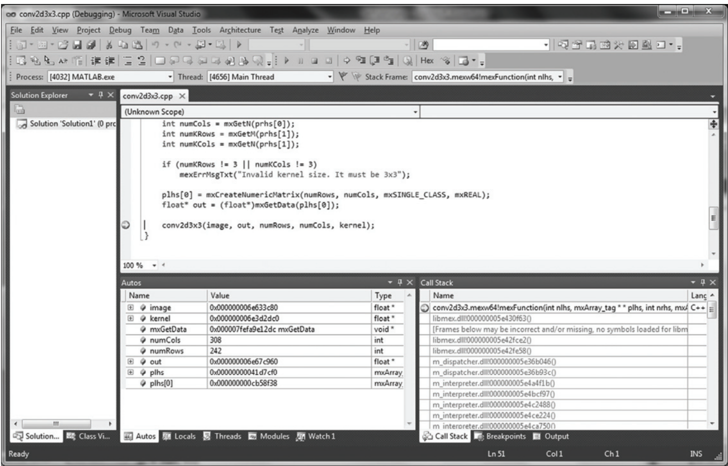


图 3.43 运行 MATLAB 主模块后自动激活的调试模式

从现在开始，你可以自由地使用 Visual Studio 中的任何调试菜单，例如利用 Step into (F11)和 Step over (F10)来跟踪变量变化。图 3.44 中的 Autos 框展示了一个范例，在这个范例中，可以通过操作 Step into (F11)和 Step over (F10)来观察变量的值。

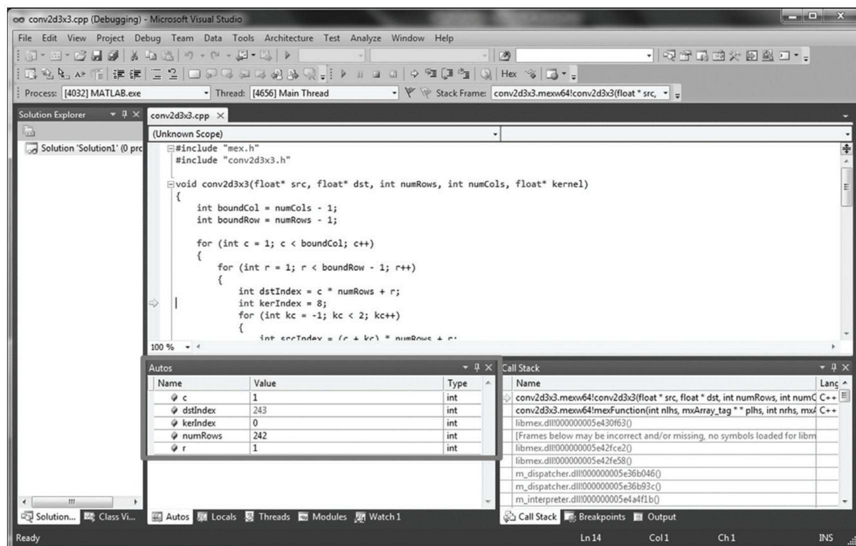


图 3.44　带有 MATLAB 的 Microsoft Visual Studio 调试器下的调试工具示例

第 4 章　利用 c-mex 进行 CUDA 编程

4.1　本章学习目标

了解使用的工具，可以更好地利用它，最大化发挥其功能。在编写 c-mex 函数时，必须准确了解数据在 MATLAB 和 c-mex 函数之间的传递过程。如果不准确了解这一过程，很可能白白浪费宝贵的时间。同样，在 CUDA 中，如果很好地了解 GPU 知识，就可以最大化发挥 CUDA 的功能。

本章中，你可以了解到以下内容：

- c-mex 中的存储布局。
- GPU 硬件基础知识。
- CUDA 中的线程分组。

4.2　c-mex 中的存储布局

理解输入数据的存储布局对于 c-mex 编程的成功至关重要。当输入数组传递给子例行程序 mexFunction 时，会得到一个输入实参数组，作为形参 prhs。每个输入实参都是 mxArray 的一个指针。为简单起见，暂时只重点关注二维数组。举个例子，使用下面的 MATLAB 预处理宏和应用程序编程接口（Application Program Interface，API）来获得数组信息：

mxIsSingle	确定 mxArray 数据是否表示为单精度浮点数
mxGetM	获得行的数量
mxGetN	获得列的数量
mxGetData	获得数据的指针

一旦得到了数据指针后，将该指针用作剩余部分的普通 C/C++指针。因此，我们必须明确地知道 MATLAB 数据在存储器空间是如何存储的。

4.2.1　按列存储

遵从 FORTRAN 语言习惯，所有 MATLAB 数据都是按列的顺序存储的。在按列顺序中，每一列和其他列都是在存储位置连续依次存储的。为了对按列顺序存

储有更好的理解，考虑下面 3×4 的矩阵：

$$\boldsymbol{A}=\begin{bmatrix}1 & 2 & 3 & 4\\5 & 6 & 7 & 8\\9 & 10 & 11 & 12\end{bmatrix}$$

在存储器空间中，每个元素存储如下：

数据偏移	0	1	2	3	4	5	6	7	8	9	10	11
数据	1	5	9	2	6	10	3	7	11	4	8	12

在 MATLAB 命令窗口中，输入如下代码：

```
A = [1 2 3 4; 5 6 7 8; 9 10 11 12]
```

在 c-mex 函数中，这个矩阵作为 prhs 中的一个实参进行传递。假设这是第一个输入元素，我们把这个矩阵作为子例行程序的一块单精度数据，然后会得到如下指针：

```
float* ptr = (float*)mxGetData(prhs[0]);
```

在 C/C++语言中，索引从零开始[㊀]，使用指针运算可以访问每一个元素。为了得到数值 10，可以采用如下语句：

```
ptr[5]
or
*(ptr + 5)
```

通常，为了访问 $M\times N$ 二维矩阵中第 m 行第 n 列的元素，采用如下方式计算指针偏移量和进行访问：

```
ptr[ (n * M) + m ]
    or
*(ptr + (n * M) + m)
```

对于 $M\times N\times P$ 的三维数组，按如下方式访问第 m 行第 n 列第 p 页的元素：

```
ptr[ ( p * N + n) * M + m ]
    or
*(ptr + ((p * N + n) * M + m))
```

如你所见，当顺序访问存储器中的数据时，行指针比列指针变化更快。

下面的范例展示了 c-mex 函数中二维数组的存储位置。一个简单数组传递给 c-mex 函数，对于单精度和字节数据类型，只需在 MATLAB 命令窗口中打印每个数组元素和它们相应的内存地址。

㊀ 注意：在 MATLAB 中，矩阵的索引是从 1 开始，然而在 C/C++中则是从 0 开始。

对于单精度数据类型：

```
// column_major_order_single.cpp

  #include "mex.h"

  void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
  *prhs[])
  {
       if (!mxIsSingle(prhs[0]))
           mexErrMsgTxt("input vector data type must be single");

      int rows = (int)mxGetM(prhs[0]);
      int cols = (int)mxGetN(prhs[0]);
     int totalElements = rows * cols;

      float* A = (float*)mxGetData(prhs[0]);

      for (int i = 0; i < totalElements; ++i)
           mexPrintf("%f at %p\n", A[i], A + i);
  }

>> mex column_major_order_single.cpp
>> column_major_order_single(single([1 2 3 4; 5 6 7 8; 9 10 11 12]))
 1.000000 at 0x128406240
 5.000000 at 0x128406244
 9.000000 at 0x128406248
 2.000000 at 0x12840624c
 6.000000 at 0x128406250
10.000000 at 0x128406254
 3.000000 at 0x128406258
 7.000000 at 0x12840625c
11.000000 at 0x128406260
 4.000000 at 0x128406264
 8.000000 at 0x128406268
12.000000 at 0x12840626c
```

（你可以用范例 column_major_single.m 进行测试。）

注意，每个数据元素占用 4 字节，这是典型的单精度数据字节大小。

下面的范例给出了单字节数据类型，即 8 位无符号整数：

```
// column_major_order_unit8.cpp

#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
```

```
    if (!mxIsUint8(prhs[0]))
        mexErrMsgTxt("input vector data type must be single");

    int rows = (int)mxGetM(prhs[0]);
    int cols = (int)mxGetN(prhs[0]);
    int totalElements = rows * cols;

    unsigned char* A = (unsigned char*)mxGetData(prhs[0]);

    for (int i = 0; i < totalElements; ++i)
        mexPrintf("%f at %p\n", A[i], A + i);
}

>> mex column_major_order_uint8.cpp
>> column_major_order_uint8(uint8([1 2 3 4;5 6 7 8; 9 10 11 12]))
 1 at 0x128408960
 5 at 0x128408961
 9 at 0x128408962
 2 at 0x128408963
 6 at 0x128408964
10 at 0x128408965
 3 at 0x128408966
 7 at 0x128408967
11 at 0x128408968
 4 at 0x128408969
 8 at 0x12840896a
12 at 0x12840896b
```

（你可以用范例 column_major_uint8.m 进行测试。）

可能你会注意到，对于单精度数据类型，内存地址一次增加 4 字节，而对于 8 位无符号整数，内存地址则一次增加 1 字节。

4.2.2　按行存储

按行顺序存储恰好与按列顺序存储相反。考虑相同的范例矩阵：

```
a = [1 2 3 4; 5 6 7 8; 9 10 11 12]
```

数据偏移	0	1	2	3	4	5	6	7	8	9	10	11
数据	1	2	3	4	5	6	7	8	9	10	11	12

数值 10 以如下方式访问：

```
ptr[9]
or
*(ptr + 9)
```

对于二维数组，$M \times N$ 矩阵中第 m 行第 n 列的元素以如下方式访问：

```
ptr[(m*N)+n]
or
*(ptr+(m*N)+n)
```

对于三维数组，$M \times N \times P$ 矩阵中第 m 行第 n 列第 p 页的元素以如下方式访问：

```
ptr[(p*M+m)*N+n]
or
*(ptr+(p*M+m)*N+n)
```

这里，列指针比行指针变化更快。

当混合使用 C/C++和 MATLAB 时，我们要特别注意，输入数据是如何存储和索引的。例如，许多 C/C++图像库是按行顺序存储图像，而 MATLAB 中传送来的数据则是按列顺序存储的。在 CUDA 中，访问线程是基于按行顺序的。另外一个非常重要，必须牢记的就是索引方案。C/C++是以零为基准进行索引，而 MATLAB 访问数组则是以 1 为基准进行索引。

4.2.3 c-mex 中复数的存储布局

我们会经常在很多地方遇到复数。尤其是在图像和信号处理中，数据可能是复数，也可能是实数，或者两者兼有。通过算法可以将实数变成复数，或者将复数变成实数。因此，当在 c-mex 函数中接收来自 MATLAB 的数据，或者传送数据给 MATLAB 时，要确保了解 MATLAB 在存储器空间中是如何打包复数的。

当在存储器中存储复数时，可以先放实数部分再放虚数部分，或者反过来。通常，先存复数的实数部分。例如，复数 C=3+7i 存储如下：

数据偏移	0	1
数据	3	7

接下来快速验证一下，MATLAB 如何存储复数：

```
// TestComplex1.cpp

#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    if (!mxIsComplex(prhs[0]))
        mexErrMsgTxt("input data must be complex");
    if (!mxIsSingle(prhs[0]))
        mexErrMsgTxt("input data must be single");

    float* pReal = (float*)mxGetPr(prhs[0]);
    float* pImag = (float*)mxGetPi(prhs[0]);

    mexPrintf("%p: %f\n", pReal, *pReal);
```

```
    mexPrintf("%p: %f\n", pImag, *pImag);
}
```

编译这一段 c-mex 测试代码，并用样本数运行这段代码：

```
>> mex TestComplex1.cpp
>> TestComplex1(single(4+5i))
000000006F2FBB80: 4.000000
000000006F2FC940: 5.000000
```

（你可以用范例 TestComplex_first.m 进行测试。）

样本数的实部存储在 0x6F2FBB80，而虚部存储在 0x6F2FC940。如你所见，它们彼此并不相邻。事实上，MATLAB 是在分开的两个数组中存储复数的实部和虚部。因此，MATLAB 提供了两个函数来访问这些数组，mxGetPr(...) 提供访问实数数组的指针，mxGetPi(...) 提供访问虚数数组的指针。对测试程序做一点修改，观察当数据是一组复数时会发生什么情况：

```
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    if (!mxIsComplex(prhs[0]))
        mexErrMsgTxt("input data must be complex");
    if (!mxIsSingle(prhs[0]))
        mexErrMsgTxt("input data must be single");

    float* pReal = (float*)mxGetPr(prhs[0]);
    float* pImag = (float*)mxGetPi(prhs[0]);
    int m = (int)mxGetM(prhs[0]);
    int n = (int)mxGetN(prhs[0]);
    int numElems = (m >= n) ? m : n;

    for (int i = 0; i < numElems; ++i, ++pReal, ++pImag)
    {
        mexPrintf("Real = %f @%p\t", *pReal, pReal + i);
        mexPrintf("Imag = %f @%p\n", *pImag, pImag + i);
    }
}
```

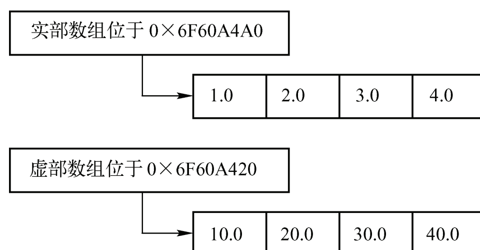


图 4.1　c-mex 中的复数数组

当编译和运行样本复数数组时，可以得到如下数据：

```
>> mex TestComplex.cpp
>> TestComplex(single([1+10i, 2+20i, 3+30i, 4+40i]))
Real=1.000000 @000000006F60A4A0   Imag=10.000000 @000000006F60A420
Real=2.000000 @000000006F60A4A8   Imag=20.000000 @000000006F60A428
Real=3.000000 @000000006F60A4B0   Imag=30.000000 @000000006F60A430
Real=4.000000 @000000006F60A4B8   Imag=40.000000 @000000006F60A438
```

（你可以用范例 TestComplex_second.m 进行测试）

可以清楚地看到，存储复数的实部和虚部为两个不同的数组，如图 4.1 所示。

如果输入数据为二维复数，会收到两个二维数组：一个实部的二维数组和一个虚部的二维数组。需牢记，它们是按列顺序存储的。

4.3　逻辑编程模型

当编写 CUDA 程序（.cu 文件）时，对于需要运行在 GPU 上的函数，我们采用特殊的语法<<<...>>> 。第 2 章有一个范例 AddVectors.cu。

	// AddVectors.cu in Chapter 2 with C-mex #include "AddVectors.h"
Kernel	__global__ void addVectorsKernel(float* A, float* B, float* C, int size) { int i = blockIdx.x; if (i >= size) return; C[i] = A[i] + B[i]; }
Code for the Host (CPU)	void addVectors(float* A, float* B, float* C, int size) { float *devPtrA = 0, *devPtrB = 0, *devPtrC = 0; cudaMalloc(&devPtrA, sizeof(float) * size); cudaMalloc(&devPtrB, sizeof(float) * size); cudaMalloc(&devPtrC, sizeof(float) * size); cudaMemcpy(devPtrA, A, sizeof(float) * size, cudaMemcpyHostToDevice); cudaMemcpy(devPtrB, B, sizeof(float) * size, cudaMemcpyHostToDevice);
Calling kernel	addVectorsKernel<<<size, 1>>>(devPtrA, devPtrB, devPtrC, size);
	cudaMemcpy(C, devPtrC, sizeof(float) * size, cudaMemcpyDeviceToHost); cudaFree(devPtrA); cudaFree(devPtrB); cudaFree(devPtrC); }

这部分运行在 GPU 上的代码称为核函数（kernel）。当定义核函数时，必须明确指定核函数运行所需的线程总数。大致说来，相同的核函数将在多个线程上并行运行。所以必须告诉 GPU 需要多少个线程并行运行。根据需要并行运算的工作量，估计所需线程数。这些线程在逻辑上组成线程块，一组线程块组成线程网格。这一逻辑分组信息置于<<<...>>>中传递给 GPU：

```
Kernel_Function <<< gridSize, blockSize >>> (parameter1, parameter2,....)
```

线程网格的尺寸由线程块的数量指定，线程块的尺寸由线程的数量指定。例如，如果线程网格尺寸是 10，一个线程网格中就有 10 个线程块。如果线程块的尺寸是 5，一个线程块中就有 5 个线程。因此，如果线程网格尺寸为 10，线程块的尺寸为 5，就表明总共有 50 个线程。

比较 CPU 和 GPU 代码。在 CPU 代码中，operation_at_cpu(..) 中的加法在循环迭代中进行。迭代条件由 for-loop 中的 (i=0; i<N; i++)明确地设置。而 GPU 代码中相应部分执行加法，则是通过若干线程并行完成的。并行化条件由 CUDA 内置设备变量（blockDim、blockIdx 和 threadIdx）的组合来设定。当调用核函数时，设备变量如 blockDim、blockIdx 和 threadIdx 受限于<<<...>>>中线程网格尺寸和线程块尺寸。当调用 GPU 设备时，实际的并行运行由 GPU 设备自动激活。blockDim、blockIdx 和 threadIdx 的内容将在 4.5 节介绍。

CPU code	**GPU code**
void operation_at_cpu(float *a, float val, int N) { for (int i = 0; i < N; i++){ a[i] = a[i] + val; } }	__global__ void operation_at_gpu(float *a, float val, int N) { int i = blockIdx.x * blockDim.x + threadIdx.x; if (i < N){ a[i] = a[i] + val; } }
void mexFunction(...) { ... operation_at_cpu(a, val, N); }	void mexFunction(...) { ... dim3 blockSize (number_Of_Threads); dim3 gridSize(ceil(N / (float) number_Of_Threads)); operation_at_gpu <<< gridSize, blockSize >>> (a, val, N); }

在 CUDA 中，需要告诉 GPU 需要并行运行的线程数，以及它们在逻辑上如

何分成线程块和线程网格。

假设做一个简单的 3×4 矩阵加法：

$$\boldsymbol{A}=\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix},\quad B=\begin{bmatrix} m & n & o & p \\ q & r & s & t \\ u & v & w & x \end{bmatrix}$$

$$\boldsymbol{A}+\boldsymbol{B}=\begin{bmatrix} a+m & b+n & c+o & d+p \\ e+q & f+r & g+s & h+t \\ i+u & j+v & k+w & l+x \end{bmatrix}$$

很容易地确定，这 12 个独立的加法可以并行运行，并且可以通知 GPU 用不同的方法将这些线程组成线程网格和线程块，如图 4.2 所示。

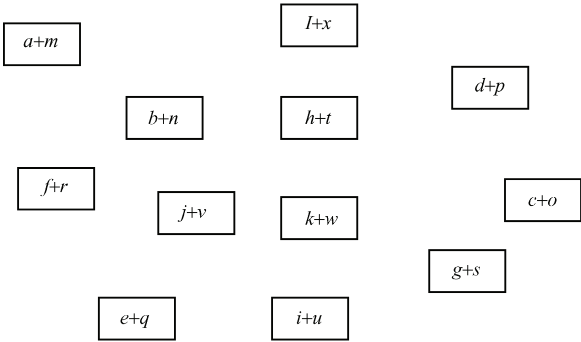


图 4.2　12 个独立的加法可以并行运行

4.3.1　逻辑分组 1

12 个独立的加法可以有各种分组方式，即一个线程网格有 12 个线程块，每个线程块仅有 1 个线程，如图 4.3 所示。

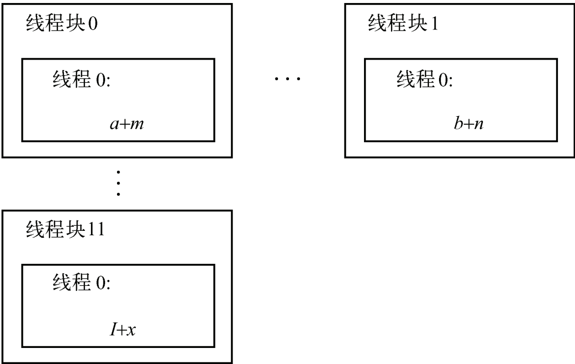


图 4.3　每线程块有 1 个线程的逻辑分组

4.3.2　逻辑分组 2

另一种线程分组方法是 1 个线程网格有 6 个线程块，每个线程块有 2 个线程，如图 4.4 所示。



图 4.4　每个线程块有两个线程的逻辑分组

4.3.3　逻辑分组 3

CUDA 还提供了二维或三维上的分组方法。可以把这些线程分成每个线程网格中有 2×2 个线程块，而每个线程块中有 2×3 个线程。这样总共产生了 24 个线程，这是所需线程数的 2 倍，如图 4.5 所示。

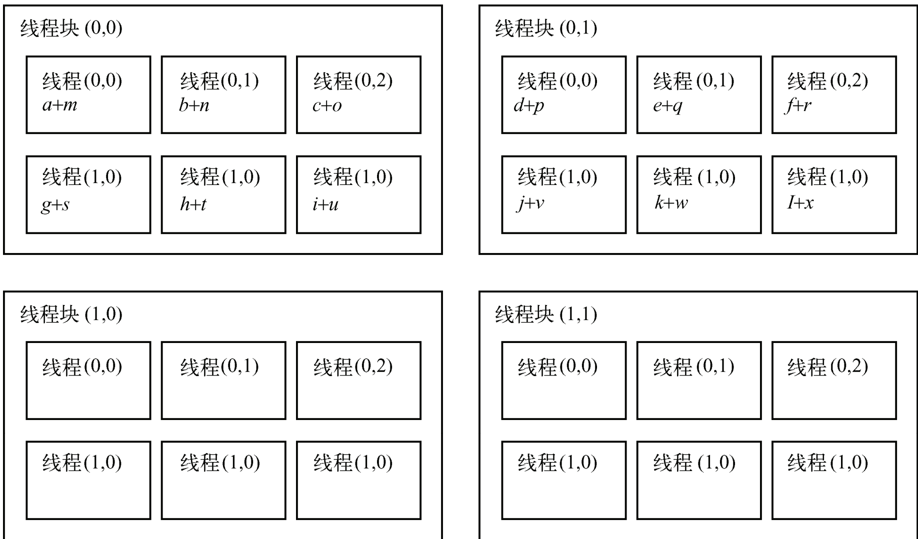


图 4.5　二维逻辑分组

注意，有一些线程块会有空线程，即无需进行任何计算。通知 GPU 共有 24 个线程，但是只有 12 个线程实际工作。在核函数中，通过检查线程索引来跳过这些空线程。

很自然地，你会问应该如何给线程分组。要回答这个问题，让我们简单地看看 GPU 硬件的细节。

4.4 GPU 简单介绍

第 2 章中，利用了 CUDA 进行二维卷积算法，测得的时间性能分析结果非常令人失望。让我们静下心来仔细想想，最初的 CUDA 编程到底出了什么问题。可能你已经想到，CUDA 编程成功与否与理解 GPU 架构有密切的关系。接下来简要地回顾一下 GPU 是如何设计的，以便能够更好地使用 GPU。

4.4.1 数据并行

与 CPU 不同，GPU 以高吞吐量为设计目标，例如 GPU 可以对一帧图像的数百万个像素进行同样的三角网格化处理。实际上，GPU 可以同时运行数千个线程，比 CPU 的线程数要多得多，这为数值并行处理开辟了新的方法。基于这一原因，GPU 因其具有同时运行数千个线程的能力，非常自然地适合实现数据的并行处理。实现数据并行化是第一步。在图像卷积的范例中，每个像素能够独立处理与其他像素并行。如果图像的大小为 10×10 像素，可以给每个线程分配一个像素，并让所有 100 个线程同时进行处理。

在硬件方面，GPU 采用单指令多数据（Single Instruction, Multiple Data，SIMD）模式。在这一模式下，一条指令可以处理多个不同的数据。与 CPU 相比，GPU 拥有更多的内核和寄存器，可以处理成千上万的轻量级线程。GPU 在硬件层面上调度这些线程。在 GPU 中，上下文切换开销几乎为零，而在 CPU 中，上下文切换的开销是十分巨大的。

4.4.2 流处理器

GPU 的基本单元是流处理器（Streaming Processor，SP）。流处理器具有从存储器中获取单指令，并在不同数据集上并行执行这条指令的能力。在一个硬件上，单个 GPU 包含数以万计的流处理器。

4.4.3 流处理器簇

多个流处理器组成一个流处理器簇（Steaming Multiprocessor，SM）。流处理器簇提供数以千计的寄存器、线程调度器，和一组流处理器之间的共享存储器。

通过配备这些组件和多个流处理器，流处理器簇可以管理线程调度、上下文切换、数据获取以及分配给多个线程的缓存。

4.4.4　线程束

流处理器簇特以 32 个并行线程为一组来调度线程，这 32 个并行线程叫做“线程束”。同一线程束中的所有线程共享从存储器中获取的单条指令。每个线程束中所有线程按照统一步调运行。只有当前线程束中所有的线程都完成工作时，流处理器簇才会处理下一个线程束。例如，如果一个线程用 10s 完成工作，而其他线程只用了 1s，则流处理器簇需要等待 9s 再开始下一个线程束。

总体来说，GPU 接收到许多线程网格。线程网格中的线程块将分配给流处理器簇。然后流处理器簇把线程分配给流处理器，并以线程束为单元管理这些线程，如图 4.6 和图 4.7 所示。线程以线程束的方式进行处理。实际同时运行的线程数受流处理器数量限制。流处理器簇负责在线程束中调度线程。

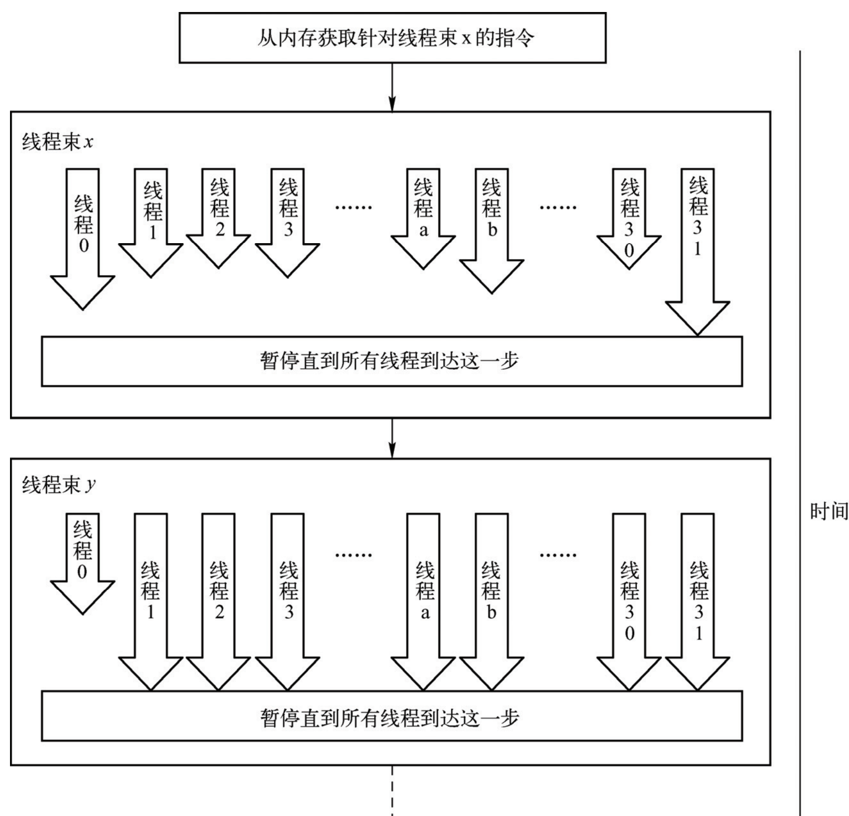


图 4.6　GPU 获得线程束中线程的一个指令。只有线程束中的所有线程都完成这一指令后，它才会移动到下一个线程束

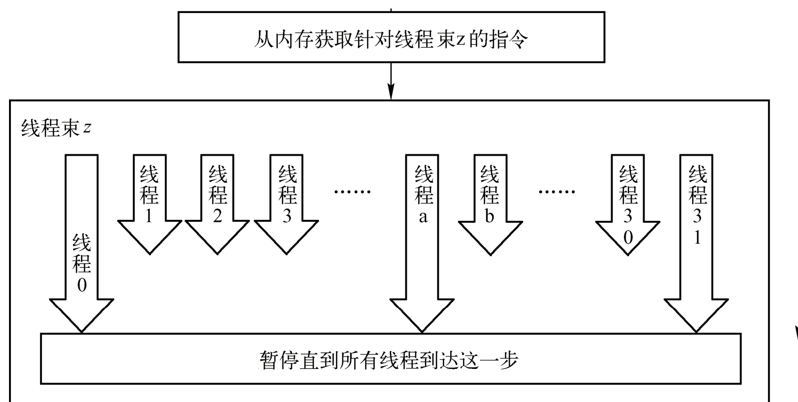


图 4.6 GPU 获得线程束中线程的一个指令。只有线程束中的所有线程都完成这一指令后，它才会移动到下一个线程束（续）

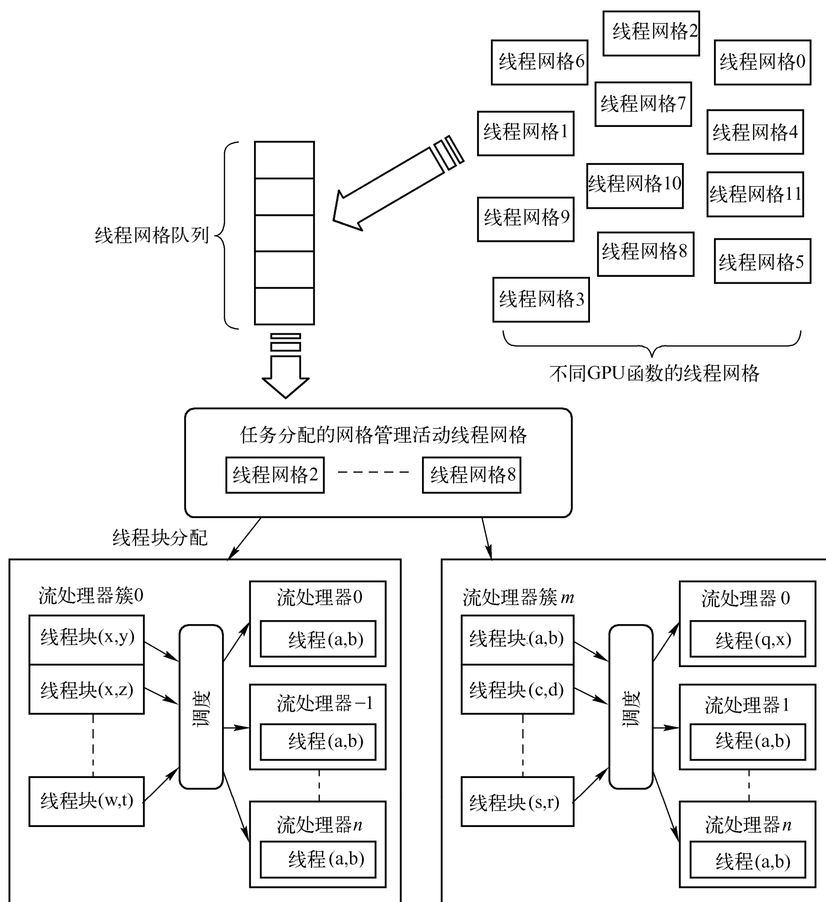


图 4.7 在 GPU 中，线程网格、线程块和线程是如何调度的

4.4.5　存储器

GPU 中主要有三类可用的存储器。第一类为全局存储器。GPU 和 CPU 都可以访问这一存储器空间。通常，在 CPU 侧，首先将全局存储器分配给 GPU，然后将 CPU 存储器中的数据复制到 GPU。所有线程网格和线程也可以访问这些存储器空间。特别地，我们使用 CUDA runtime API 调用诸如 cudaMalloc 和 cudaMemcpy 的函数来分配内存，并从 GPU 复制数据到 CPU，或者从 CPU 复制数据到 GPU。

第二类为共享存储器。并不是所有的线程都有权访问共享存储器，只有在同一线程块中的线程才有权访问。然而访问速度比全局存储器要快很多。可以用关键词 __shared__ 来分配这种类型的存储器。如果数值需要，且一个线程块中的线程可以共享这些数值，就可以把这些线程的数据存放到共享存储器中，这样可以减少存储器访问的时间开销。

第三类存储器只对每个线程可用，这就是使用寄存器的单线程本地存储器。与 CPU 不同，GPU 每个 SM 上有数以千计的寄存器。这些寄存器专门针对每个线程，而且线程上下文切换开销几乎为零。在核函数中，声明数据为本地变量，即可获准使用寄存器。然而，核函数中本地存储器的容量是十分有限的，基于这个原因，必须确保不会滥用。当寄存器不能满足需求时，GPU 就会将它们放在适用于本地线程的本地存储器中，但是这不会像访问寄存器那样快。

图 4.8 显示了 GPU 存储器概况。

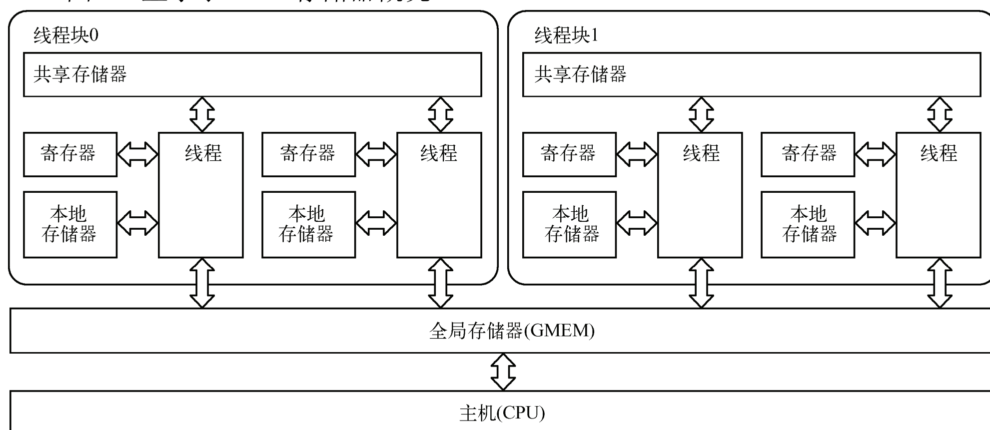


图 4.8　存储器空间的高层概况

4.5　第一种初级方法的分析

如第 2 章所示，我们第一次利用 CUDA 实现二维卷积比采用 MATLAB 内置函数 conv2 实现二维卷积要慢得多。了解了 GPU 内部结构后，现在我们开始解决这个问题。在第一种方法中，盲目地给每个线程网格分配一个线程块。每个线程

块分配一个线程，这里给出了需要运行的线程总数，以如下方式声明核函数：

```
dim gridSize(numRows, numCols)
conv2MexCuda<<<gridSize, 1>>>
```

在这个声明中，有 numRows×numCols 个线程块，每个线程块有一个线程。因此，总共有 numRows×numCols 个线程。记住，在 MATLAB 中，数据是按列顺序传送的。所以先分配 numRows，然后再分配 numCols。在图 4.9 中，水平地移动，数据线程块映像的第一个指针是行数，而第二个指针是列数。在输入图像中，线程块中的每个线程处理一个像素点。

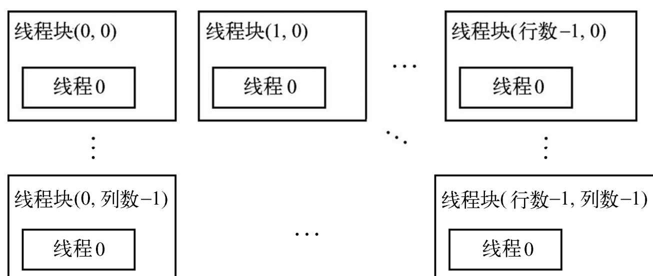


图 4.9　将每个像素分配给一个线程块中的一个线程

每个线程网格中的线程块分配给 SM，而每个 SM 在线程束中调度线程。在初级方法中，当一个 SM 处理线程时，每个线程束中只有一个线程在运行。然而一个 SM 在一个线程束中可以处理 32 个线程。由于一个线程束中只有一个线程，SM 中的多数 SP 都处于闲置，GPU 资源利用极不充分。

再次运行 NVIDIA Visual Profiler，观察当给每个线程块分配一个线程时它是如何执行的，如图 4.10 所示。

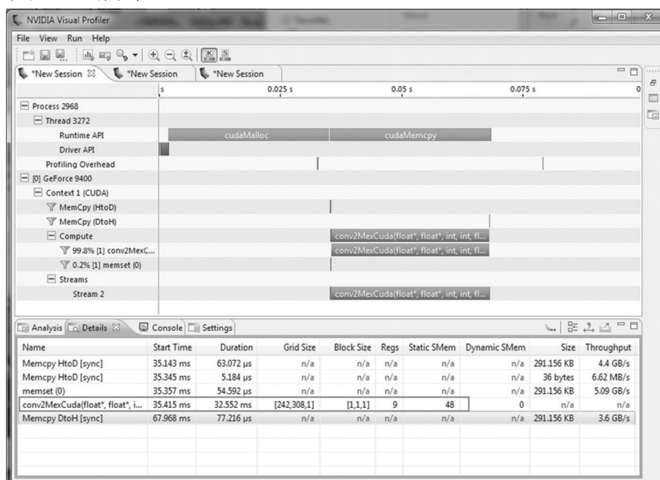


图 4.10　使用初级方法的 Visual Profiler

正如在 Details 栏中看到的那样，当线程网格大小与图像大小相同时，核函数本身需要大约 33ms。每个线程处理一个像素。利用第一种方法，甚至使得基于 CUDA 的卷积比利用 MATLAB 内置函数的卷积速度更慢。因此，在这种情况下，没有得到 GPU 加速的优势。在 4.5.1 节中，将试着调整线程网格和线程块的大小，以便在 SM 中可以利用线程束中的其他线程。

4.5.1　优化方案 A：线程块

在优化步骤中，我们通过调整线程网格和线程块的大小来进行优化。首先，确定一个块的大小为 16×16，每个块共有 256 个线程，很明显，它比线程束的尺寸更大。基于线程块和图像的大小，可以确定线程网格的大小。在第一种方法中，每个线程块中只利用了其中一个线程。所以本方法通过给每个线程块分配更多的线程，利用线程块中的其他线程。

每个新的线程块中包含 256 个以二维方式排列的线程，如图 4.11 所示。线程块中的每个线程对应处理输入图像的每个像素。图 4.12 为 MATLAB 的原始输入图像。

0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0	…	14,0	15,0
0,1	1,1	2,1	3,1	4,1	5,1	6,1	7,1	…	14,1	15,1
0,2	1,2	2,2	3,2	4,2	5,2	6,2	7,2	…	14,2	15,2
			⋮					⋱	⋮	
0,14	1,14	2,14	3,14	4,14	5,14	6,14	7,14	…	14,14	15,14
0,15	1,15	2,15	3,15	4,15	5,15	6,15	7,15	…	14,15	15,15

图 4.11　大小为 16×16 的线程块



图 4.12　MATLAB 中显示的输入图像

出于演示的目的，首先将输入图像进行转置。这是因为来自 MATLAB 的输入图像是按列顺序存储的，然而在 C/C++和 CUDA 中，图像是按行顺序存储的。然后，把线程块置于输入图像的顶部，看看这个原始图像是如何分为线程块和线程的，如图 4.13 所示。

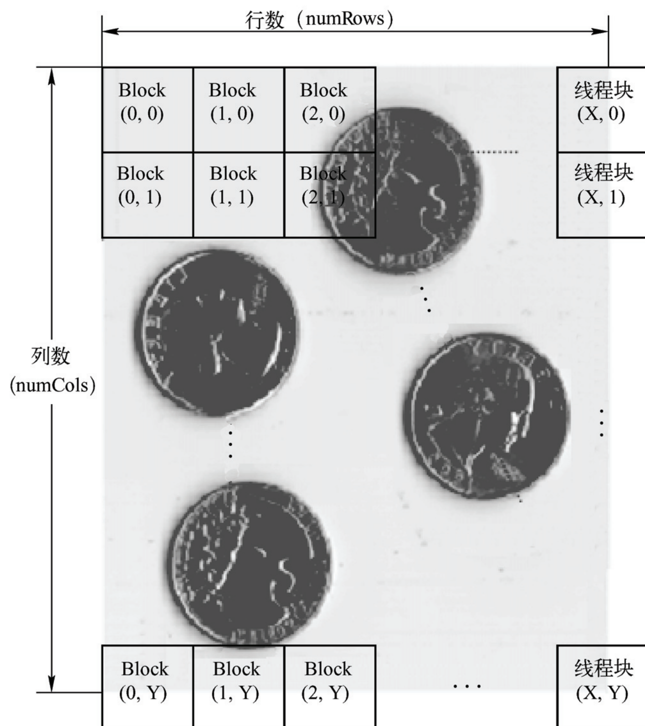


图 4.13　由线程块覆盖的输入图像

给定输入图像和线程块的大小，可以确定在水平和垂直方向上分别有多少线程块。注意到线程块覆盖的整个区域比输入图像大。这是因为所有的线程块应该是相同大小的，而图像的大小可能不是线程块大小的整数倍。因此，要使线程块覆盖的整个区域比输入图像大。

```
dim3 blockSize(16, 16);
dim3 gridSize((numRows +15) / blockSize.x, (numCols + 15) / blockSize.y);
```

这一段代码展示了如何计算线程块的最小数量，这个数量必须是 16 的整数倍，且大于等于图片的大小。这里介绍两个 dim3 类型的变量 blockSize 和 gridSize。

现在，通过线程指针和所分配线程块大小（每个像素对应一个线程），可以访问图像中的每个像素。当调用 CUDA 核函数时，可以知道哪个线程块中的哪个线程

正在进行处理。在核函数中，使用以下 CUDA 的全局变量来找到确切的像素位置。

- blockDim：线程块的大小（例子中是 16×16）。
- blockIdx：　线程网格中线程块的索引。
- threadIdx：线程块中线程的索引。

每个特定像素的位置可以用图 4.14 所示的变量计算。请再次注意，图像的行是水平表示的，而图像的列是垂直呈现的，因为在 MATLAB 中，图像是按列顺序存储的。

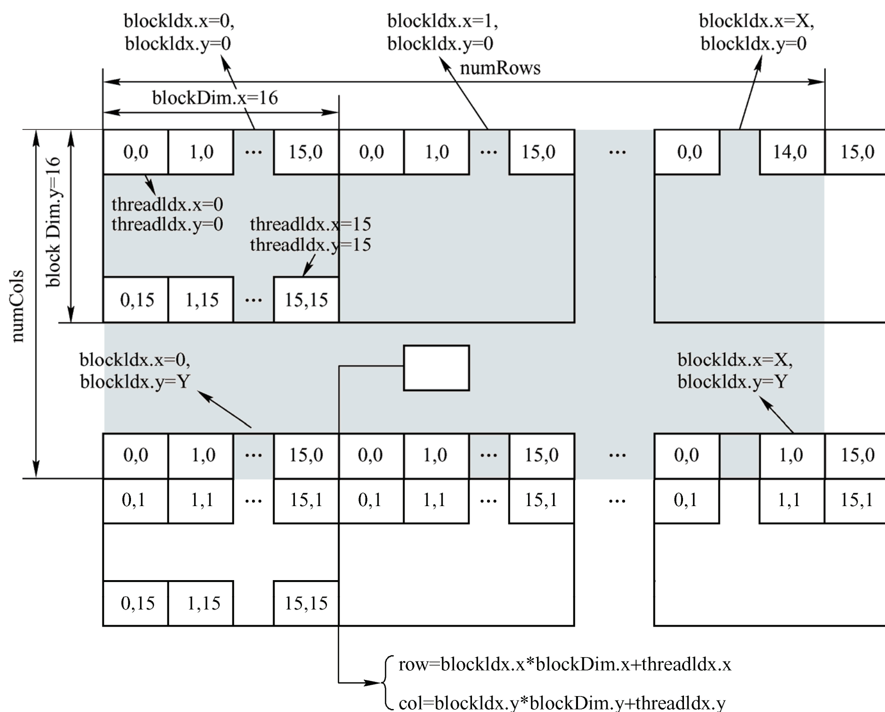


图 4.14　分配给图像的线程块和线程

对于分配了图像以外像素的线程将直接返回，不进行任何处理。在本例中，也忽略了卷积操作的边界区域。在核函数中，通过如下方式检查操作是否在边界内进行：

```
int row = blockIdx.x * blockDim.x + threadIdx.x;
if (row < 1 || row > numRows - 1)
    return;

int col = blockIdx.y * blockDim.y + threadIdx.y;
if (col < 1 || col > numCols - 1)
    return;
```

现在创建一个新文件，并保存为 conv2MexOptA.cu。

```
#include "conv2Mex.h"
__global__ void conv2MexCuda(float* src,
                             float* dst,
                             int numRows,
                             int numCols,
                             float* mask)
{
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < 1 || row > numRows - 1)
        return;

    int col = blockIdx.y * blockDim.y + threadIdx.y;
    if (col < 1 || col > numCols - 1)
        return;

    int dstIndex = col * numRows + row;
    dst[dstIndex] = 0;
    int mskIndex = 3 * 3 - 1;
    for (int kc = -1; kc < 2; kc++)
    {
        int srcIndex = (col + kc) * numRows + row;
        for (int kr = -1; kr < 2; kr++)
        {
            dst[dstIndex] += mask[mskIndex--] * src[srcIndex + kr];
        }
    }
}

void conv2Mex(float* src, float* dst, int numRows, int numCols, float* msk)
{
    int totalPixels = numRows * numCols;
    float *deviceSrc, *deviceMsk, *deviceDst;

    cudaMalloc(&deviceSrc, sizeof(float) * totalPixels);
    cudaMalloc(&deviceDst, sizeof(float) * totalPixels);
    cudaMalloc(&deviceMsk, sizeof(float) * 3 * 3);

    cudaMemcpy(deviceSrc, src, sizeof(float) * totalPixels,
    cudaMemcpyHostToDevice);
    cudaMemcpy(deviceMsk, msk, sizeof(float) * 3 * 3,
    cudaMemcpyHostToDevice);
    cudaMemset(deviceDst, 0, sizeof(float) * totalPixels);

    dim3 blockSize(16, 16);
    dim3 gridSize((numRows +15) / blockSize.x, (numCols + 15) / blockSize.y);
    conv2MexCuda <<< gridSize, blockSize >>> (deviceSrc,
                                              deviceDst,
```

```
                                                numRows,
                                                numCols,
                                                deviceMsk);

    cudaMemcpy(dst, deviceDst, sizeof(float) * totalPixels,
    cudaMemcpyDeviceToHost);

    cudaFree(deviceSrc);
    cudaFree(deviceDst);
    cudaFree(deviceMsk);
}
```

在这个范例代码中，创建了一个新的核函数并以如下方式调用它：

```
conv2MexCuda<<<gridSize, blockSize>>>(deviceSrc,
                                      deviceDst,
                                      numRows,
                                      numCols,
                                      deviceMsk);
```

我们向核函数传送新的线程块和线程大小。与之前每个线程块中只处理一个像素的方法不同，这里将图像重组成 16×16 线程块，但是，与之前方法相同，每个线程执行 3×3 掩膜乘法和加法。

然后，利用 NVIDIA Visual Profiler 分析这个新方法。时间分析结果显示，新方法有了巨大的改善，运算速度相比之前的方法提高了 30 多倍，如图 4.15 所示。

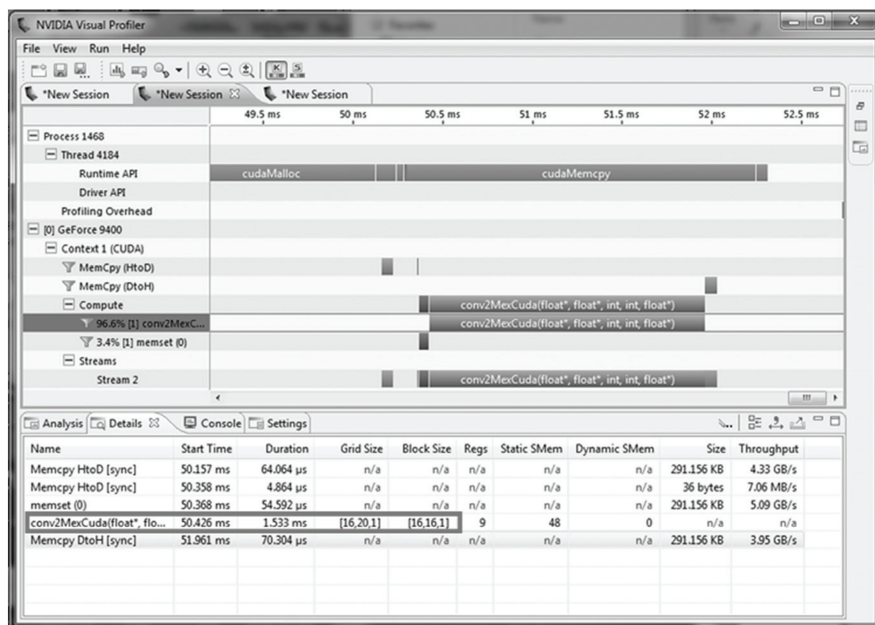


图 4.15　采用新线程块大小优化后的卷积

4.5.2 优化方案 B

在这个优化方案中，其他地方保持不变，只是在 CUDA 核函数中引入共享存储器。在优化方案 A 中，每个像素计算卷积时，要反复使用相同的 9 个掩膜值。在每次进行掩膜乘法时，都要从全局存储器中查询这些值。从 GPU 全局存储器中获得这些值，比从共享存储器或者寄存器更耗时。所以，可以将掩膜值存放到共享存储器中，使得全局存储器的访问开销最小化。

在核函数中，做如下声明：

```
__shared__ float sharedMask[9];
```

这条指令将在 GPU 中创建一个共享存储器，并且一个线程块中的所有线程都可以共享。因此，线程块中所有 256 个线程都可以使用这一共享存储器。然后，最初的 9 个线程用掩膜值填满共享存储器。当准备好用于存放卷积掩膜值的共享存储器后，就可以进行实际计算了。为了在填满共享存储器之前，确保没有其他线程继续计算，调用以下函数：

```
__syncthreads();
```

现在，创建并保存一个新文件 convMexOptB.cu，如下：

```
#include "conv2Mex.h"

__global__ void conv2MexCuda(float* src,
                             float* dst,
                             int numRows,
                             int numCols,
                             float* mask)
{
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < 1 || row > numRows - 1 || col < 1 || col > numCols - 1)
        return;

    __shared__ float sharedMask[9];

    if (threadIdx.x < 9)
    {
        sharedMask[threadIdx.x] = mask[threadIdx.x];
    }
    __syncthreads();

    int dstIndex = col * numRows + row;
    dst[dstIndex] = 0;
```

```
    int mskIndex = 8;
    for (int kc = -1; kc < 2; kc++)
    {
        int srcIndex = (col + kc) * numRows + row;
        for (int kr = -1; kr < 2; kr++)
        {
            dst[dstIndex] += sharedMask[mskIndex--] * src[srcIndex + kr];
        }
    }
}

void conv2Mex(float* src, float* dst, int numRows, int numCols, float* msk)
{
    int totalPixels = numRows * numCols;
    float *deviceSrc, *deviceMsk, *deviceDst;

    cudaMalloc(&deviceSrc, sizeof(float) * totalPixels);
    cudaMalloc(&deviceDst, sizeof(float) * totalPixels);
    cudaMalloc(&deviceMsk, sizeof(float) * 3 * 3);

    cudaMemcpy(deviceSrc, src, sizeof(float) * totalPixels,
    cudaMemcpyHostToDevice);
    cudaMemcpy(deviceMsk, msk, sizeof(float) * 3 * 3,
    cudaMemcpyHostToDevice);
    cudaMemset(deviceDst, 0, sizeof(float) * totalPixels);

    const int size = 16;
    dim3 blockSize(size, size);
    dim3 gridSize((numRows + size - 1) / blockSize.x,
                  (numCols + size - 1) / blockSize.y);

    conv2MexCuda<<<gridSize, blockSize>>>(deviceSrc,
                                          deviceDst,
                                          numRows,
                                          numCols,
                                          deviceMsk);

    cudaMemcpy(dst, deviceDst, sizeof(float) * totalPixels,
    cudaMemcpyDeviceToHost);

    cudaFree(deviceSrc);
    cudaFree(deviceDst);
    cudaFree(deviceMsk);
}
```

前面已经介绍了共享存储器。因为核函数中的数值对于所有像素是一样的，因此可以把核函数中的数值放到共享存储器中，从而使 16×16 个线程可以从 L1

缓存中共享相同的核函数值，而不是从全局存储器中获取这些值。

用 NVIDIA Visual Profiler 编译并运行上述代码，得到如图 4.16 所示的结果。相比于第一步的优化，性能得到了一定的改善。

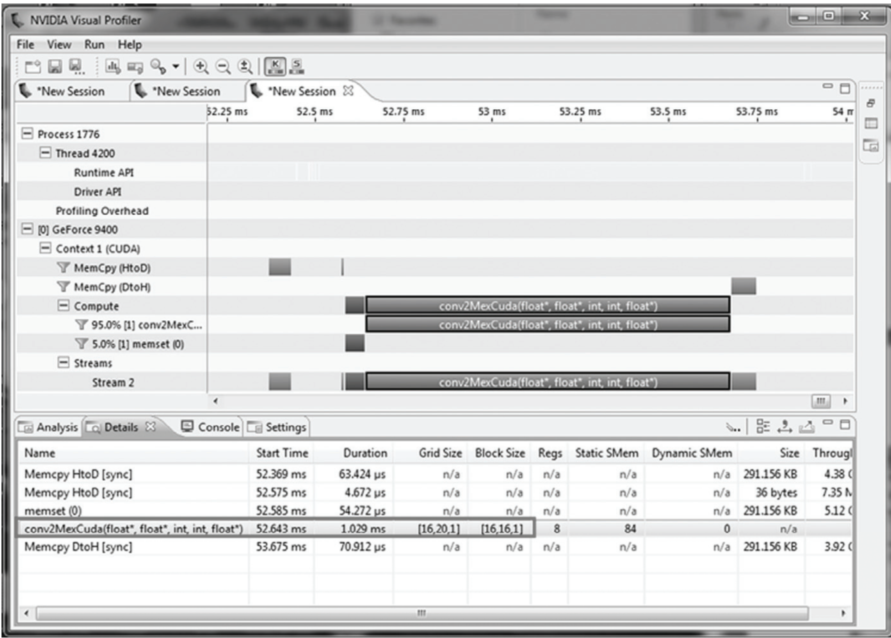


图 4.16　利用共享存储器进行优化

4.5.3　总结

可以看到，相比于第一种初级方法，整体性能确实有了很大的改善。但是，当考虑端到端函数调用时，这种方法仍落后于 MATLAB 的内置函数 conv2。是什么原因使得整体速度变慢？注意到，无论何时调用函数，一定要运行 cudaMemcpy。在采用这种方法时，要从主机的存储器复制数据到 GPU 的存储器，或者从 GPU 的存储器复制数据到主机存储器。虽然实际上 GPU 内部的卷积运算速度非常快，但这一过程开销极大，所以，要在 GPU 内部做尽可能多的计算，并尽可能减少主机与 GPU 设备之间的数据传递。

第 5 章　MATLAB 与并行计算工具箱

5.1　本章学习目标

MATLAB 并行计算工具箱为并行处理提供了有用的工具。该工具箱提供多种并行处理方式，例如联网的多台计算机、多核计算机的多个内核、集群计算以及 GPU 并行处理。本书更多关注并行计算工具箱中的 GPU 部分。并行计算工具箱的优点之一是可以使用 GPU 而无需直接进行 CUDA 编程或者 c-mex 编程。不过，安装并行计算工具箱需要额外付费。本章将讨论以下内容：

- GPU 处理 MATLAB 内置函数。
- GPU 处理 MATLAB 非内置函数。
- 并行任务处理。
- 并行数据处理。
- 无 c-mex 的 CUDA 文件直接使用。

5.2　GPU 处理 MATLAB 内置函数

MATLAB 并行计算工具箱仅支持 1.3 或者更高版本的 CUDA。在 MATLAB 命令窗口中使用 gpuDevice 指令，检查 CUDA 版本以及其他 GPU 相关信息，如图 5.1 所示。

```
>> gpuDevice

ans =

  CUDADevice with properties:

                      Name: 'GeForce GT 640M LE'
                     Index: 1
         ComputeCapability: '3.0'
            SupportsDouble: 1
             DriverVersion: 5
            ToolkitVersion: 5
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
               MaxGridSize: [2.1475e+09 65535 65535]
                 SIMDWidth: 32
               TotalMemory: 1.0737e+09
                FreeMemory: 972619776
       MultiprocessorCount: 2
              ClockRateKHz: 570000
               ComputeMode: 'Default'
      GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 1
          CanMapHostMemory: 1
           DeviceSupported: 1
            DeviceSelected: 1
```

图 5.1　通过 gpuDevice 指令得到 GPU 信息

在图 5.1 中，Name 显示安装在计算机中的 GPU 硬件卡，ComputeCapability 表明安装在计算机系统中的 CUDA 库软件版本（本例为 CUDA 3.0）。如果无法通过 gpuDevice 指令得到信息，那么进行下一步之前，需要先检查 GPU 硬件状态或 CUDA 软件安装。

现在，让我们开始在 MATLAB 中为 GPU 准备数据。正如第 3 章提到的，由于 GPU 通过 PCI 总线连接到 CPU，因此使用 GPU 计算时，始终需要考虑主机（CPU 和主存储器）与设备（GPU 和 GPU 存储器）之间的数据传送。用如下的 gpuArray 指令，将数据从 MATLAB 工作空间（位于主存储器上）传输到 GPU 设备存储器：

```
%test_gpuArray.m
A = 1:0.01:50000;                % About 5 million elements
A_gpu = gpuArray(A);             % transfer data to GPU device memory
tic; B = fft(A); toc             % FFT on CPU
tic; B_gpu = fft(A_gpu); toc     % FFT on GPU
B_from_gpu = gather(B_gpu);      % return data back to MATLAB workspace
```

本例中，向量 A 通过 gpuArray 指令传输至 GPU 设备存储器。也可以直接给 GPU 提供数据，如下所示：

```
>> A_gpu = gpuArray(1:0.01:50000);
```

这里，fft 在 CPU 和 GPU 两者上都进行计算。要注意的是，对于 CPU 和 GPU，我们使用了相同的函数名（fft），不过由于输入数据的不同（A 是主存储器上的数据，而 A_gpu 是 GPU 存储器上的数据），计算指令将分别在 CPU 和 GPU 上运行。可以在 MATLAB 工作空间查看差异，如图 5.2 所示。

Workspace

Name	Value	Min	Max
A	<1x4999901 double>	<Too many element...	<Too many element...
A_gpu	<1x4999901 gpuArray>	<Too many element...	<Too many element...
B	<1x4999901 complex double>	<Too many element...	<Too many element...
B_from_gpu	<1x4999901 complex double>	<Too many element...	<Too many element...
B_gpu	<1x4999901 gpuArray>	<Too many element...	<Too many element...

图 5.2 MATLAB 工作空间中，主存储器上的数据 A 显示为 double，而 GPU 设备存储器上的数据 A_gpu 显示为 gpuArray

用 gather 指令将数据从 GPU 设备存储器送回 MATLAB 工作空间。上述代码的运行结果如下所示：

```
>> test_gpuArray
Elapsed time is 1.725417 seconds.
Elapsed time is 0.644574 seconds.
```

本例的运行结果令人振奋，GPU 处理约 500 万个数据的运行时间约为 CPU 的三分之一。不过，来看看下面的代码：

```
%test_gpuArray_small.m

A = 1:2:500;                        % data size = 250

A_gpu = gpuArray(A);                % transfer to GPU device memory

tic;  B = fft(A);  toc
tic;  B_gpu = fft(A_gpu);  toc

B_from_gpu = gather(B_gpu);
>> test_gpuArray_small
Elapsed time is 0.000240 seconds.
Elapsed time is 0.000655 seconds.
```

在这个例子中，数据量很小，在 GPU 上运行并没能获得速度提升。这符合前面第 3 章提到的，只有对于密集计算的情形，转换为 GPU 代码才有希望获得加速。

可以通过 methods('gpuArray')指令来查看支持 gpuArray 的 MATLAB 内置函数：

```
>> methods('gpuArray')
```

这样，你可以得到支持 gpuArray 的内置函数列表。这意味着并不是所有的 MATLAB 内置函数都完全支持 GPU 处理，不过在 MATLAB 并行计算工具箱的每个新版本中，支持 GPU 的函数都在不断增加。下表为 MATLAB R2013a 版支持 GPU 的内置函数：

```
Methods for class gpuArray:

abs              ezsurfc            normest
acos             feather            not
acosh            fft                num2str
acot             fft2               numel
acoth            fftfilt            or
acsc             fftn               padarray
acsch            fill               pareto
all              fill3              pcolor
and              filter             permute
any              filter2            pie
applylut         find               pie3
area             fix                plot
arrayfun         floor              plot3
```

asec
asech
asin
asinh
atan
atan2
atanh
bar
bar3
bar3h
barh
beta
betaln
bitand
bitcmp
bitget
bitor
bitset
bitshift
bitxor
bsxfun
bwlookup
cast
cat
cconv
ceil
chol
circshift
clabel
classUnderlying
comet
comet3
compass
complex
cond
coneplot
conj
contour
contour3
contourc
contourf
contourslice
conv
conv2
convn
cos
cosh
cot
coth
cov
fplot
fprintf
full
gamma
gammaln
gather
ge
gpuArray
gt
hist
horzcat
hypot
ifft
ifft2
ifftn
imag
image
imagesc
imbothat
imclose
imdilate
imerode
imfilter
imopen
imrotate
imshow
imtophat
ind2sub
int16
int2str
int32
int64
int8
interp1
interpstreamspeed
inv
ipermute
isa
isempty
isequal
isequaln
isequalwithequalnans
isfinite
isfloat
isinf
isinteger
islogical
ismember
isnan
isnumeric
plotmatrix
plotyy
plus
polar
pow2
power
prod
qr
quiver
quiver3
rdivide
real
reallog
realpow
realsqrt
reducepatch
reducevolume
rem
repmat
reshape
ribbon
rose
round
scatter3
sec
sech
semilogx
semilogy
shiftdim
shrinkfaces
sign
sin
single
sinh
size
slice
smooth3
sort
sprintf
spy
sqrt
stairs
stem
stem3
stream2
stream3
streamline
streamparticles
streamribbon
streamslice

```
csc              isocaps          streamtube
csch             isocolors        sub2ind
ctranspose       isonormals       subsasgn
cumprod          isosurface       subsindex
cumsum           isreal           subsref
curl             issorted         subvolume
det              issparse         sum
diag             ldivide          surf
diff             le               surfc
disp             length           surfl
display          log              svd
divergence       log10            tan
dot              log1p            tanh
double           log2             times
eig              logical          transpose
end              loglog           tril
eps              lt               trimesh
eq               lu               trisurf
erf              mat2str          triu
erfc             max              uint16
erfcinv          mesh             uint32
erfcx            meshc            uint64
erfinv           meshgrid         uint8
errorbar         meshz            uminus
existsOnGPU      min              uplus
exp              minus            var
expm1            mldivide         vertcat
ezcontour        mod              vissuite
ezcontourf       mpower           volumebounds
ezgraph3         mrdivide         voronoi
ezmesh           mtimes           waterfall
ezmeshc          ndgrid           xcorr
ezplot           ndims            xor
ezplot3          ne
ezpolar          nnz
ezsurf           norm

Static methods:

colon            logspace         randn
eye              nan              true
false            ones             zeros
inf              rand
linspace         rand
```

当使用 GPU 处理大量数据时，应进行数据验证。在 CPU 处理时，如果试图运行的数据超出了当前操作系统预先分配的内存，那么操作系统会通过与硬盘进行存储交换，自动地处理内存不足。虽然存储交换会减慢计算速度，但是无需担心数据丢失。然而在 GPU 处理时，GPU 设备不会与硬盘进行存储

交换，因此使用者应该验证 GPU 上的数据有没有超出 GPU 设备内存容量的上限。

看看下面的例子：

```
%verify_gpuData.m

gpuDevice

A = rand(10000);        % 10000×10000 double on main memory
A_gpu = gpuArray(A);    % 10000×10000 double on GPU memory

gpuDevice

B = rand(10000);        % Another 10000×10000 double on main memory
B_gpu = gpuArray(B);    % Another 10000×10000 double on GPU memory
```

```
>> verify_gpuData

ans =
    CUDADevice with properties:

                      Name: 'GeForce GT 640M LE'
                     Index: 1
         ComputeCapability: '3.0'
            SupportsDouble: 1
             DriverVersion: 5
            ToolkitVersion: 5
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
               MaxGridSize: [2.1475e+09 65535 65535]
                 SIMDWidth: 32
               TotalMemory: 1.0737e+09
                FreeMemory: 973668352
       MultiprocessorCount: 2
              ClockRateKHz: 570000
               ComputeMode: 'Default'
      GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 1
          CanMapHostMemory: 1
           DeviceSupported: 1
            DeviceSelected: 1

ans =
    CUDADevice with properties:

                      Name: 'GeForce GT 640M LE'
                     Index: 1
         ComputeCapability: '3.0'
            SupportsDouble: 1
             DriverVersion: 5
            ToolkitVersion: 5
```

```
      MaxThreadsPerBlock:   1024
        MaxShmemPerBlock:   49152
      MaxThreadBlockSize:   [1024 1024 64]
             MaxGridSize:   [2.1475e+09 65535 65535]
               SIMDWidth:   32
             TotalMemory:   1.0737e+09
              FreeMemory:   173604864
     MultiprocessorCount:   2
            ClockRateKHz:   570000
             ComputeMode:   'Default'
    GPUOverlapsTransfers:   1
  KernelExecutionTimeout:   1
        CanMapHostMemory:   1
         DeviceSupported:   1
          DeviceSelected:   1

Error using gpuArray
Out of memory on device. To view more detail about
available memory on the GPU, use 'gpuDevice()'. If the
problem persists, reset the GPU by calling
'gpuDevice(1)'.

Error in verify_gpuData (line 11)
B_gpu = gpuArray(B); % Another 10000×10000 double on
GPU memory
```

在一切开始前，首先用 gpuDevice 进行查询，GPU 设备的空闲内存为 973 668 352 字节。在运行 A_gpu = gpuArray(A)后，GPU 设备的空闲内存减至 173 604 864 字节。通过指令 B= rand(10000)，大矩阵 B 安全地分配到主机的主存储器上，不过在运行 B_gpu = gpuArray(B)后，会有一个来自 GPU 端的错误信息。这种情况下，应该通过 clear 指令或者重置全部 GPU 设备内存，清除一些 GPU 数据，给新的 GPU 变量留出额外的空间，如下所示：

```
>> clear A_gpu
```

或

```
>> g = gpuDevice(1);
>> reset(g);
```

5.3　GPU 处理非内置 MATLAB 函数

现在来寻找将 GPU 用于自行编写的 MATLAB 函数的方法。可以采用 arrayfun 函数，在 GPU 上运行自行编写的 MATLAB 代码。arrayfun 通过元素操

作将自行编写的函数用于每个数据元素。实际上，arrayfun 函数不是 GPU 专用函数，而是 GPU 支持函数。因此，可以在 5.2 节 MATLAB 并行计算工具箱中的 GPU 支持函数列表内找到 arrayfun 函数。

使用 arrayfun 意味着给并行的 GPU 运算做了单独的调用，这个调用执行了全部的运算，而不是为各个分离的 GPU 最优运算产生很多个调用。对于存储在 GPU 存储器上的 gpuArray 数据，自行编写的 MATLAB 函数通过 arrayfun 可以在 GPU 上执行，并且将输出存在 GPU 存储器中。

可采用如下形式使用 Arrayfun：

```
result = arrayfun(@myFunction, arg1, arg2, ...);
```

其中，arg1 和 arg2 是 myFunction 的输入实参，这些输入实参应为用于 GPU 处理的 gpuArray 数据。myFunction 应为标量（也就是按元素计算）运算，因此不支持向量和矩阵计算。myFunction 的输出是 result，也同样存储在 GPU 存储器中。

让我们看看下面这个简单的例子：

```
% test_arrayfun.m

a = gpuArray(1:0.1:10);
b = gpuArray(2:0.1:11);
c = gpuArray(3:0.1:12);
d = gpuArray(4:0.1:13);

gpu_rlt = arrayfun(@myFunc,
a,b,c,d);

rlt = gather(gpu_rlt);
```

```
% my own element-wise MATLAB function
% myFunc.m

function out = myFunc(a, b, c, d)
out = b / (a * d * sin(c));
```

代码 myFunc.m 有 4 个输入实参(a, b, c, d)，均通过 test_arrayfun.m 中的 gpuArray 存储在 GPU 存储器上。非内置函数 myFunc.m 只由标量运算构成，标量运算可以在 GPU 内很容易地实现并行化。通过 test_arrayfun.m 中的 arrayfun 函数，我们实现了在 GPU 上运行 myFunc 函数。

尽管 arrayfun 的输入函数只能进行元素级运算，但是这并不意味着只允许进行简单运算。反而，通过 arrayfun，输入函数可以在 GPU 上进行任意标量运算和流程控制（例如 for-loop、while-loop、break、if 等）：

```
% user's scalar MATLAB function

function [out, count] = myGoodFunc(a, b, maxIter)

    count = 0;
    out = 0;
    amp = abs(a^2 + b^2);
```

```
    while(count <= maxIter)&(amp < 1.5)

        out = (a^3 - b/a);
        count = count + 1;
    end
% scalar_arrayfun.m

A = gpuArray.rand(40,40);
B = gpuArray.rand(40,40);

max_Iter = 3000;

[gpu_rlt, gpu_count] = arrayfun(@myGoodFunc, A, B, max_Iter);

rlt = gather(gpu_rlt);
count = gather(gpu_count);
```

然而，需要矩阵元素索引的正则矩阵和向量运算无法通过 arrayfun 在 GPU 上运行。

```
% test_arrayfun2.m

A = gpuArray.rand(2,5,4);
B = gpuArray.rand(2,5,4);

gpu_rlt = arrayfun(@myFunc2, A, B);

rlt = gather(gpu_rlt);
```

```
% myFunc2.m

function out = myFunc2(A, B)

a1 = A(1,1,1);  % Matrix indexing

A(1,1,1) = a1 * B(1,2,1) - 3;
out = B ./ A;
```

由于 myFunc2.m 中的矩阵索引运算（matrix indexing operation），test_arrayfun2.m 将导致如下错误信息：

```
>> test_arrayfun2
Error using gpuArray/arrayfun
Indexing is not supported. error at line: 6

Error in test_arrayfun2 (line 6)
gpu_rlt = arrayfun(@myFunc2, A, B);
```

5.4　并行任务处理

5.4.1　MATLAB worker

并行计算工具箱为并行“任务”处理（parfor，将在 5.4.2 节中介绍）和并行“数据”处理（spmd，将在 5.5.1 节中介绍）均提供了非常高效的方法。虽然在并行计算工具箱中，并行任务处理和并行数据处理方法更多地与 CPU 相关，而不是 GPU，然而当这些方法与 GPU 处理相结合时，可以明显提升速度。

每个 CPU 内核都具有独立的 MATLAB 会话，称之为“worker”㊀。当前的 MATLAB 版本（2013a），一台计算机可以最多拥有 12 个 worker，并且默认 worker 数目由计算机可用内核的数目确定。为了将主 MATLAB 的会话与每个 worker 各自的会话区分开，称主 MATLAB 会话为“客户端”。因此，任何主 MATLAB 会话在访问 worker 之前和之后均是客户端。对于并行处理，我们应通过指令 matlabpool 准备可用的 worker，如图 5.3 所示。

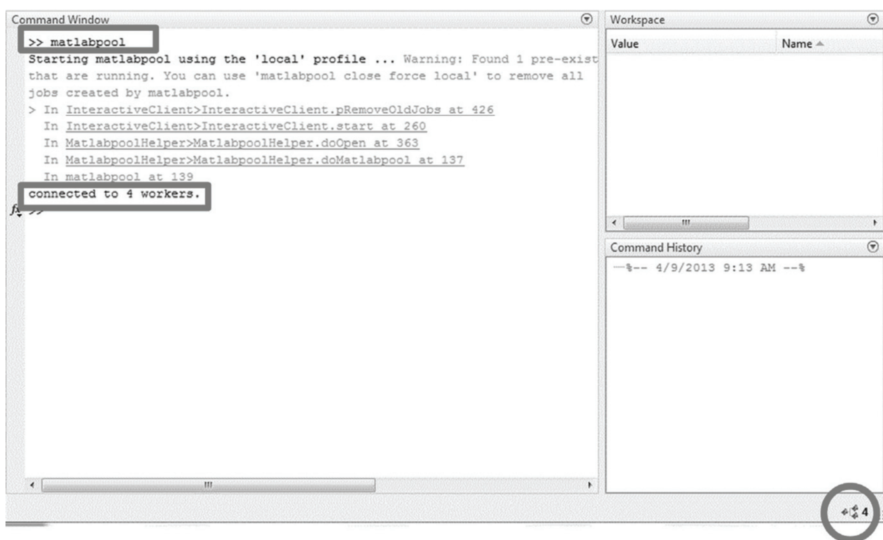


图 5.3 用 matlabpool 指令准备可用的 worker

在图 5.3 中，执行命令 matlabpool 后，能从 MATLAB 指令窗口中的 connected to 4 workers 和右下角显示的数字 4，确定默认的 worker 数目。可以通过图 5.4 和图 5.5 所示的 Manage Cluster Profile 菜单，来更改默认的 worker 数目。

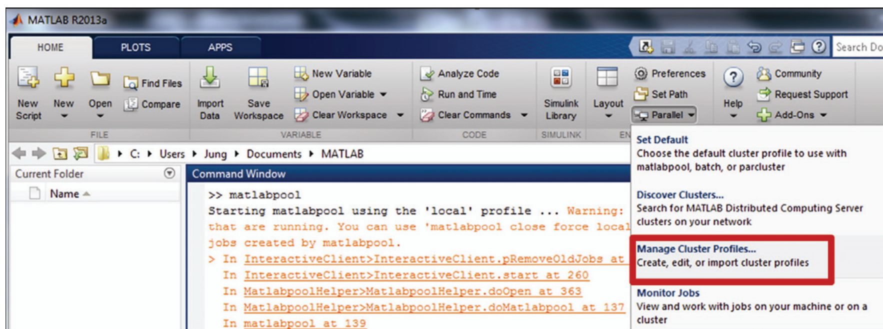


图 5.4 进入 Parallel 菜单里的 Manage Cluster Profiles…来更改默认的 worker 数目

㊀ 由于 MATLAB 分布式计算服务器产品可以让使用者在远程计算机集群上运行额外的 worker，因此称单机多核 worker 为本地 worker。简单起见，本书中只采用单机进行并行处理，所以这里的 worker 即为“本地 worker”。

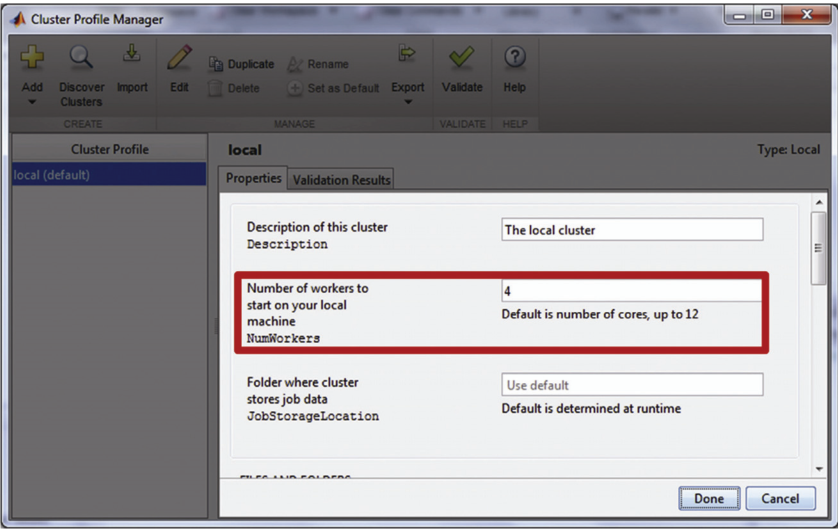


图 5.5　要修改默认 worker 数目，可在 Cluster Profile Manager 窗口中设置

如果计算机上只有 4 个内核，而要求 8 个 worker，MATLAB 会产生 8 个 worker，但是它们会共享 4 个内核。多个 worker 共享同一内核有可能提升处理速度，也有可能不提升，具体情况取决于代码。

5.4.2　parfor

使用 parfor 取代 for-loop，可以轻松实现任务并行化。在用 matlabpool 指令预留 MATLAB worker 后，使用 parfor，可以将 for-loop 之间的命令行自动地分布于各个 worker 上，如下所示：

```
for i = 1:100
    command
     ...
end
```

<table>
<tr><td rowspan="2"><pre>matlabpool

parfor i = 1:100
    command
     ...
end</pre></td><td colspan="4">如果“matlabpool”准备了4个 worker，则左侧的“parfor-loop”将隐式地分为 4 个“for-loop”命令和并行执行</td></tr>
<tr><td><pre>for i = 1:25
    command
     ...
end</pre></td><td><pre>for i = 26:50
    command
     ...
end</pre></td><td><pre>for i = 51:75
    command
     ...
end</pre></td><td><pre>for i = 76:100
    command
     ...
end</pre></td></tr>
</table>

在前面的例子中，为了解释 parfor 的概念，各个 for-loop 循环在 worker 之间精确地均匀分布（1～25、26～50、51～75 和 76～100)，但是实际任务中并不总是均匀分布的，这依赖于 CPU 上运行的其他程序。

对于 parfor 循环，不同输入数据的相同运算可以“任务独立”和“命令独立”的方式并行地执行。也就是说，当任务之间无相关性时，parfor 循环在并行任务处理中提供了很多益处。由于 parfor 循环用于并行操作，因此循环中不能包含 break 和 return 语句。尽管 parfor 循环内部允许普通的 for-loop，但是同样也不能包含另一个 parfor 循环构成循环嵌套。

下面我们通过莱布尼茨公式计算 π 值，比较普通的 for-loop 指令与 parfor 循环指令：

$$\pi = \cfrac{4}{1+\cfrac{1^2}{2+\cfrac{3^2}{2+\cfrac{5^2}{2+\dots}}}} = \sum_{n=0}^{\infty}\frac{4(-1)^n}{2n+1}$$

```
% pi_Leib.m

tic
N = 10000000;
sum = 0;

for i = 1:(N-1)

    Denom = 2 * (i-1) + 1;
    Sign = (-1)^(i-1);
    sum = sum + 4 * Sign / Denom;
end


sum
toc
```

```
%pi_Leib_parfor.m

matlabpool
tic

N = 10000000;
sum = 0;

parfor i = 1:(N-1)

    Denom = 2 * (i-1) + 1;
    Sign = (-1)^(i-1);

    sum = sum + 4 * Sign / Denom;
end
sum

toc
```

```
>> pi_Leib

sum =
     3.1416

Elapsed time is 25.535713 seconds.

>> pi_Leib_parfor

sum =
     3.1416

Elapsed time is 8.170260 seconds.
```

例子 pi_Leib_parfor.m 使用了 4 个 worker，运行时间约为使用普通 for-loop 的 pi_Leib.m 的 1/3。

5.5　并行数据处理

5.5.1　spmd

并行计算工具箱提供的 spmd（单程序多数据）是一种非常高效的“数据”并行处理方法。与 parfor 一样，spmd 通过 matlabpool 使用 MATLAB worker。这里“单程序多数据”的含义是完全相同的代码同时在多个 MATLAB worker 上处理不同的数据，从而实现并行处理。spmd 的基本语法格式如下：

```
spmd
   command
      ...
end
```

parfor 与 spmd 有所不同，首先 spmd 将数据划分成许多个小块，而 parfor 将循环分成更小的循环。因此，parfor 要求 worker 之间相互独立或不通信，而 spmd 允许不同 worker 之间相互通信和具有依赖关系，因此更加灵活。就 parfor 而言，一个循环由 MATLAB 自动划分到每个 worker 上。而在 spmd 的情况下，用户将每一个数据段明确地划分到每一个 worker 上。因而，spmd 比 parfor 指令需要更多的用户控制。由于 parfor 与 spmd 均使用 worker，因此 parfor 循环体不能包含 spmd 语句，而 spmd 语句也不能包含 parfor 循环。不过，只要 spmd 块没有自我嵌套，那么一个程序可以有多个 spmd 块。

与 parfor 不同，用户可通过使用 labindex( )和 numlabs( )明确地分配并行数据处理 spmd 语句内的指令。

指令 labindex( )返回当前 worker 的唯一性索引，而 numlabs( )返回了可用 worker 的数目。在未使用 labindex( )函数的情况下，spmd 隐式地使用每个 worker，这意味着在没有用户干预时，数据处理将自动分配给每个 worker。

```
% spmd_index.m

matlabpool

A = 1:10;
B = 11:20;

spmd
    index = labindex();
    if index = = 1
        D = A .* B;
```

```
    end
end
matlabpool close
```

上述代码在不同 worker 上执行不同运算，具体执行何种运算取决于 labindex( )的返回值。客户端中的变量（这里为 A 与 B）是 spmd 块外的全部命令行，可以被 spmd 块中的每个 worker 引用。与 MATLAB 后续版本不同，在旧版本的 MATLAB 中 spmd 块内不能修改客户端变量。上述代码运行结果如下所示：

```
>> spdm_index
>> D
D =
   Lab 1: class = double, size = [1  10]
   Lab 2: class = double, size = [1  10]

>> D{1}

ans =
     11   24   39   56   75   96   119   144   171   200

>> D{2}

ans =
     12   14   16   18   20   22   24   26   28   30
```

来自 spmd 块的结果 D，同正则矩阵或矢量形式略有不同。MATLAB 将客户端的这种形式称为“复合对象”，其引用了分配的 worker 上的变量。因此，D 有如下的形式，该形式在每个 worker 上表示不同的数据：

```
D =
   Lab 1: class = double, size = [1  10]
   Lab 2: class = double, size = [1  10]
```

为了访问每个 worker 上各自的结果，我们采用元胞数组索引的方法，使用 D{1}和 D{2}这样的花括号。D{1}为由 worker 1 计算的 D 的值。每个 worker 上的变量可以在客户端用这种花括号的方式进行修改。

表 5.1 给出的 spmd 例子中，一个客户端的两个 worker 有各自的工作空间。这里，黑体数字代表全部工作空间中最近更新的变量。从这个表格中，我们可以发现，客户端的全部指令与全部 spmd 块本身都是串行执行，但是各个 worker 上 spmd 块内的指令则是同时执行。

让我们看一下表 5.1 中的 spmd_value_modification.m 代码。首先，通过函数 Composite( )建立每个 worker 的复合对象，并由第 3 行到第 5 行，在客户端进行初始化。复合对象也可以直接在 spmd 块内建立，如例子 spmd_index.m 中的 D。其次，第一个 spmd 块中的变量 i、k、z 对于不同 worker 有不同的值，具体取值依赖

于 labindex( )，而且独立并行地进行处理。第三，spmd 块中创建的复合变量 k 的值在客户端的工作空间（第 13 行）更改，并且更改的值将用于后面的 spmd 块，不会有任何问题。

请注意从第 15 行到第 18 行的第二个 spmd 块。所有之前在第一个 spmd 块（从第 7 行到第 11 行）中使用过的变量都存储相同的数值，除了 k{2}。虽然第一个 spmd 块在第 11 行就完全结束，不过变量 k{2}在客户端的工作空间第 13 行进行了修改，并且一些客户端程序在那之后执行。但是，如果 MATLAB 函数被另一个函数中的 spmd 块在函数内部调用，那么当函数完成后，spmd 块内变量的值会丢失，这与普通的 MATLAB 函数数据一样。

表 5.1　客户端分离的工作空间与 spmd worker

Line #	Code	Client		Worker 1					Worker 2				
	spmd_value_modification.m	x	y	i	k	z	j	m	i	k	z	j	m
1	x = 5;	5	–	–	–	–	–	–	–	–	–	–	–
2	y = 6;	5	6	–	–	–	–	–	–	–	–	–	–
3	z = Composite();												
4	z{1} = 1;	5	6	–	–	1	–	–	–	–	–	–	–
5	z{2} = 2;	5	6	–	–	1	–	–	–	–	2	–	–
6													
7	spmd												
8	i = labindex();	5	6	1	–	1	–	–	2	–	2	–	–
9	k = x + i;	5	6	1	6	1	–	–	2	7	2	–	–
10	z = 10 * i;	5	6	1	6	10	–	–	2	7	20	–	–
11	end												
12													
13	k{2} = 10;	5	6	1	6	10	–	–	2	10	20	–	–
14													
15	spmd												
16	j = labindex();	5	6	1	6	10	1	–	2	10	20	2	–
17	m = k * j;	5	6	1	6	10	1	6	2	10	20	2	20
18	end												
final		5	6	1	6	10	1	6	2	10	20	2	20

5.5.2　分布式数组与同分布数组

分布式数组是使用 spmd 进行并行数据处理最有效的方法之一。分布式数组的基本概念很简单，却很强大。当需要处理一个大矩阵时，可以将矩阵分解成很多更小的子矩阵，在多个 worker 上并行处理。由于计算一个大矩阵需要大存储器以及更长的处理时间，因此在实际中，将一个大数组分散开有很大好处。下面列出的第一个分布式的例子将说明其基本操作。（本小节假设 matlabpool 已经激活。）

```
% dist_first.m

A = rand(2,8);
dA = distributed(A);

spmd

    localPart = getLocalPart(dA);
end
```

矩阵 A 的大小为 2×8。distributed(A) 将 A 中的元素以 2×2 的形式分配给每个 worker（使用 4 个 worker），如下所示：

	Worker1		Worker2		Worker3		Worker4	
	localPart{1}		localPart{2}		localPart{3}		localPart{4}	
Col:	Col1	Col2	Col3	Col4	Col5	Col6	Col7	Col8
Row1	0.3804	0.0759	0.5308	0.9340	0.5688	0.0119	0.1622	0.3112
Row2	0.5678	0.0540	0.7792	0.1299	0.4694	0.3371	0.7943	0.5285

这里，矩阵 A 中的元素默认按列分配。在更高维的矩阵中，最后一个维度用作分配的基准。通过 getLocalPart( )函数，我们可以从 worker 得到分布式矩阵的各个部分。示例的结果如下所示：

```
>> dist_first
>> A

A =
   0.3804   0.0759   0.5308   0.9340   0.5688   0.0119   0.1622   0.3112
   0.5678   0.0540   0.7792   0.1299   0.4694   0.3371   0.7943   0.5285

>> localPart{1}

ans =
     0.3804   0.0759
     0.5678   0.0540

>> localPart{2}

ans =
     0.5308   0.9340
     0.7792   0.1299
```

distributed( ) 函数尽可能均匀地分布矩阵，这意味着，在列数为奇数的情况下，最后一个 worker 有奇数列数据，如下所示：

```
% dist_second.m

A = rand(2,7)                        % odd number of column
dA = distributed(A);

spmd

    localPart = getLocalPart(dA);
    localPart(1,1)

end
>> dist_second

A =
   0.6555   0.7060   0.2769   0.0971   0.6948   0.9502   0.4387
   0.1712   0.0318   0.0462   0.8235   0.3171   0.0344   0.3816
```

```
Lab 1:

  ans =
       0.6555

Lab 2:

  ans =
       0.2769

Lab 3:

  ans =
       0.6948

Lab 4:
  ans =
       0.4387

>> localPart

localPart =
   Lab 1:  class = double, size = [2  2]
   Lab 2:  class = double, size = [2  2]
   Lab 3:  class = double, size = [2  2]
   Lab 4:  class = double, size = [2  1]
>> localPart{3}

ans =
    0.6948   0.9502
    0.3171   0.0344

>> localPart{4}

ans =
    0.4387
    0.3816
```

请注意，分布式矩阵 localPart 有它自己的本地索引，这与最初的未分配的矩阵 A 有所差别。

这里有两种方法可以将矩阵分配给多个 worker：distributed( ) 与 codistributed( )。distributed( ) 在矩阵进入 spmd 块之前进行分块，而 codistributed( ) 在 spmd 块内部对矩阵分块。由于矩阵分配的主要目的是处理由于太大而无法由单个内核处理的大矩阵，因此在客户端创建一个大矩阵，然后分配到多个 worker（使用 distributed( )）这种方法的效率较低。在许多情况下，用 codistributed( ) 直接在多个 worker 上创建分布式矩阵更好。之前使用 distributed 函数的 dist_first.m 例子可以用 codistributed 函数修改成如下的内容。虽然分配的方式改变了，但是结果和 dist_first.m 例子相同。

```
% codist_first.m

spmd
    A = rand(2,8);
    dA = codistributed(A);
    localPart = getLocalPart(dA);
end
```

这个例子依然是先构造矩阵 A，然后在 spmd 块内分配给 worker。在这种情况下，每个 worker 各自都有矩阵 A 的复本，这比 dist_first.m 的例子效率更低。为了充分利用 codistributed 函数的优点，可以用 codistributed.rand 函数初始化矩阵 A，而不是在客户端构造正则矩阵。

```
% codist_rand.m

spmd
    dA = codistributed.rand(2,8);
    localPart = getLocalPart(dA);
end
```

codist_rand.m 用 codistributed.rand( ) 函数直接在各个 worker 上以分布式的形式构造矩阵。除了 codistributed.rand( ) 函数之外，MATLAB 还提供了大量的函数，支持在各个 worker 上分布式构造矩阵，如下所示：

```
codistributed.cell          创建分布式元胞数组
codistributed.colon         分布式冒号操作
codistributed.eye           创建分布式单位阵
codistributed.false         创建分布式“否”数组
codistributed.Inf           创建分布式无穷大数组
codistributed.NaN           创建分布式非数数组
codistributed.ones          创建分布式全 1 数组
codistributed.rand          创建分布式均匀分布伪随机数数组

codistributed.randn         创建分布式正态分布随机数数组
codistributed.spalloc       给分布式稀疏矩阵分配空间
codistributed.speye         创建分布式稀疏单位矩阵
codistributed.sprand        创建分布式均匀分布伪随机数稀疏数组

codistributed.sprandn       创建分布式正态分布伪随机数稀疏数组

codistributed.true          创建分布式“真”数组
codistributed.zeros         创建分布式全零数组
```

在各个 worker 计算后，需要将同分布矩阵还原为非分布式的大矩阵时，可以使用 gather( ) 函数进行恢复：

```
% codist_gather.m

spmd
     dA = codistributed.ones(500,500);
     localPart = getLocalPart(dA);
```

```
end
A = gather(dA);
```

当用 4 个 worker 运行程序 codist_gather.m 时，在客户端通过使用 gather( )函数将 4 个 500×125 的块矩阵合并为一个 500×500 的大矩阵：

```
>> codist_gather
>> localPart

localPart =
   Lab 1: class = double, size = [500  125]
   Lab 2: class = double, size = [500  125]
   Lab 3: class = double, size = [500  125]
   Lab 4: class = double, size = [500  125]

>> size(A)

ans =
    500   500
```

5.5.3　多个 GPU 时的 worker

如前所述，并行计算工具箱内的 worker 默认共享 CPU 内核。因此，如果用户的计算机只有一个 GPU，而多个 worker 试图访问 GPU，那么将不能期望获得加速。在很多情况下，这会使整个程序比不采用 GPU 而只使用多个 worker 更慢，这是由于各个并行运行的 worker（CPU 核）传递数据给 GPU 设备必须串行进行造成的。然而，如果用户的计算机有与 CPU 核数量相当的多个 GPU，那么多个 worker 会工作得很好。在这种情况下，我们依据 GPU 设备 ID 明确地为每个 worker 分配 GPU 设备。由于安装多个 GPU 的多核计算机当前还不普通，因此本书将不涉及在多个 GPU 上运行多个 worker 的情况。

5.6　无需 c-mex 的 CUDA 文件直接使用

当计算机中没有安装并行计算工具箱时，可以采用 c-mex 文件在 GPU 上运行 CUDA 文件（.cu 文件），这部分内容在第 2 章中已经介绍过。当有并行计算工具箱时，可以直接在 GPU 上运行 CUDA 文件，而不需要 c-mex 文件。在这种情况下，我们无需编译 c-mex 文件，只需对 CUDA 文件进行 nvcc 编译即可。

我们打算对 AddVectors.cu 的例子进行微调，该例在第 2 章中用于 c-mex。这里是 AddVectors_noCmex.cu 文件。首先，AddVectors_noCmex.cu 文件只有 CUDA 内核部分（__global__ void AddVectors_noCmex( )），与第 2 章中 AddVectors.cu 不

同，该文件具有 CUDA 存储器配置和与 c-mex 接口相关的存储复制。

```
// AddVectors_noCmex.cu with Parallel Computing Toolbox

__global__ void AddVectors_noCmex(double* A, double* B)
{
       int i = threadIdx.x;
       A[i]+= B[i];
}
```

在 MATLAB 中直接使用.cu 文件的基本流程如下所示：

```
1) nvcc -ptx AddVectors_noCmex.cu
```

在 shell（DOS 或 Linux shell）内，应该在 NVIDIA CUDA 工具箱内加 -ptx 选项，使用 nvcc 编译 cu 文件生成 ptx 文件。也可以在 MATLAB 指令窗口内使用如下 system 指令运行这一 shell 指令：

```
system('nvcc -ptx AddVectors_noCmex.cu -ccbin
"C:\ProgramFiles(x86)\Microsoft Visual Studio 10.0\VC\bin"');
```

虽然我们不编译 c-mex 文件，但是仍然需要 C/C++编译器中的 nvcc 编译 cu 文件，这是因为 cu 文件具有 C/C++格式，并且 nvcc 转发全部非 CUDA 编译步骤到 C/C++编译器。如果你的系统没有指向 C/C++编译器（例如 cl.exe）的路径，接下来可以用 -ccbin 选项确定路径。关于从 NVIDIA 网站安装 nvcc 可参考第 2 章。通常，成功安装 NVIDIA CUDA 工具箱会自动创建路径指向 nvcc.exe 位置（C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\bin）。不过，如果由于某些原因没有创建指向该位置的路径，那么需要在运行前添加 addpath('C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\bin')指令：

```
2) myCu = parallel.gpu.CUDAKernel('AddVectors_noCmex.ptx','AddVectors_
noCmex.cu');
```

使用相同文件名的 ptx 与 cu 文件，由 parallel.gpu.CUDAKernel( ) 函数生成 CUDA 核函数。当 cu 文件由多个函数组成时，可以按照下面的方法指定入口点名称：

```
myCu = parallel.gpu.CUDAKernel('AddVectors_noCmex.ptx', 'AddVectors_
noCmex','AddVectors_noCmex.cu');
```

```
3) CuOut = feval(myCu, A, B)
```

通过 CUDA 对象 AddVectors_noCmex 的函数句柄 myCu，有两个输入实参（矩阵 A 和 B）的 feval 函数运行。下面的 nvcc_noCmex.m 文件展示了编译 cu 文件、生成 CUDA 对象和使用测试输入运行的全部代码：

```
% nvcc_noCmex.m

% Use the following when MS Visual Studio 2010 or later
system('nvcc -ptx AddVectors_noCmex.cu -ccbin "C:\Program Files (x86)
\Microsoft Visual Studio 10.0\VC\bin"');

myCu = parallel.gpu.CUDAKernel('AddVectors_noCmex.ptx','AddVectors_
noCmex.cu');

A = single([1 2 3 4 5 6 7 8 9 10]);

B = single([10 9 8 7 6 5 4 3 2 1]);

N = length(A)
myCu.ThreadBlockSize = N

CuOut = feval(myCu, A, B)
```

当运行这一代码时，有如下结果：

```
>> nvcc_noCmex
AddVectors_noCmex.cu
tmpxft_00001328_00000000-5_AddVectors_noCmex.cudafe1.gpu
tmpxft_00001328_00000000-10_AddVectors_noCmex.cudafe2.gpu

N =
   10

myCu =
      parallel.gpu.CUDAKernel handle
      Package: parallel.gpu

      Properties:
          ThreadBlockSize:   [10 1 1]
        MaxThreadsPerBlock:   1024
                  GridSize:   [1 1]
          SharedMemorySize:   0
                EntryPoint:   '_Z17AddVectors_noCmexPdS_'
        MaxNumLHSArguments:   2
           NumRHSArguments:   2
             ArgumentTypes:   {'inout double vector' 'inout double
                                vector'}

Methods, Events, Superclasses

CuOut =
       11   11   11   11   11   11   11   11   11   11
```

通过 MATLAB 指令 myCu.ThreadBlockSize = N（这里 N=10 为向量 A 的长度），ThreadBlockSize 设为[10 1 1]。ThreadBlockSize 控制 AddVectors_noCmex.cu 里的 i = threadIdx.x，在 GPU 中实现对 10 个元素进行并行加法，结果为 10 个数据元素均为 11。如果把 myCu.ThreadBlockSize 的值都改为 8，如下所示，最后两个元素没有正确求和：

```
% nvcc_noCmex2.m

system('nvcc -ptx AddVectors_noCmex.cu -ccbin "C:\Program Files (x86)
\Microsoft Visual Studio 10.0\VC\bin"');
myCu = parallel.gpu.CUDAKernel('AddVectors_noCmex.ptx', 'AddVectors_
noCmex.cu');

A = single([1 2 3 4 5 6 7 8 9 10]);
B = single([10 9 8 7 6 5 4 3 2 1]);

N = length(A)
myCu.ThreadBlockSize = N-2

CuOut = feval(myCu, A, B)
>> nvcc_noCmex2

N =
   10

myCu =
      parallel.gpu.CUDAKernel handle
      Package: parallel.gpu

      Properties:
           ThreadBlockSize:   [8 1 1]
         MaxThreadsPerBlock:   1024
                  GridSize:   [1 1]
          SharedMemorySize:   0
                EntryPoint:   '_Z17AddVectors_noCmexPdS_'
         MaxNumLHSArguments:   2
           NumRHSArguments:   2
             ArgumentTypes:   {'inout double vector' 'inout double
                                vector'}

Methods, Events, Superclasses

CuOut =
        11   11   11   11   11   11   11   11   9   10
```

除了使用 c-mex 接口，AddVectors.cu 与 AddVectors_noCmex.cu 的主要区别是 CUDA 核函数实参的数目：

```
// AddVectors_noCmex.cu with Parallel Computing Toolbox

__global__ void AddVectors_noCmex(double* A, double* B)
{
       int i = threadIdx.x;

       A[i] + = B[i];
}


// AddVectors.cu in Chapter 2 with C-mex

__global__ void addVectorsKernel(float* A, float* B, float* C, int size)
```

```
{
        int i = blockIdx.x;
        if (i >= size)
           return;

        C[i] = A[i] + B[i];
}
```

在第 2 章中的 AddVectors.cu 例子中，c-mex 的 CUDA 核函数（addVectorsKernel( )）在 c-mex 运行后被其他函数调用，因此用户可以自由地规定实参的数量和顺序。不过，MATLAB 函数 feval( ) 通过 parallel.gpu.CUDAKernel 直接使用的 CUDA 核函数（addVectorsKernel( )），则应该遵循严格的规则。当调用 [output1, output2] = feval(CUDA_Kernel, input1, input2, input3) 时，input1、input2 和 input3 应该与 CUDA_Kernel 函数的输入实参一致。如果有两个以上输入实参，那么前两个输入实参用作输出实参；如果有两个或者更少的输入实参，那么仅第一个输入实参用作输出实参。可以通过 CUDA 核函数句柄来核实 CUDA 核函数的属性：

```
>> myCu

myCu =
      parallel.gpu.CUDAKernel handle
      Package: parallel.gpu

      Properties:
           ThreadBlockSize:   [10 1 1]
        MaxThreadsPerBlock:   1024
                  GridSize:   [1 1]
          SharedMemorySize:   0
                EntryPoint:   '_Z17AddVectors_noCmexPdS_'
        MaxNumLHSArguments:   2
           NumRHSArguments:   2
             ArgumentTypes:   {'inout double vector' 'inout double
                                vector'}

Methods, Events, Superclasses
```

这里，可以看到 ArgumentTypes 里有两个 inout 的描述。inout 表明了这两个实参既可以用于输入，也可以用于输出。下面看一个很简单的例子：

```
// Simple_noCmex.cu with Parallel Computing Toolbox

__global__ void Simple_noCmex(double* A, double val)
{
       int i = threadIdx.x;

       A[i] + = val;
}


% simple_noCmex.m
```

```
system('nvcc -ptx Simple_noCmex.cu -ccbin "C:\Program Files (x86)
\Microsoft Visual Studio 10.0\VC\bin"');

handleCu = parallel.gpu.CUDAKernel('Simple_noCmex.ptx',
'Simple_noCmex.cu');

A = single([1 2 3 4 5 6 7 8 9 10]);
handleCu.ThreadBlockSize = length(A);

CuOut = feval(handleCu, A, 5.0)
>> simple_noCmex
Simple_noCmex.cu
tmpxft_00001bdc_00000000-5_Simple_noCmex.cudafe1.gpu
tmpxft_00001bdc_00000000-10_Simple_noCmex.cudafe2.gpu

CuOut =
       6   7   8   9   10   11   12   13   14   15

>> handleCu

handleCu =
          parallel.gpu.CUDAKernel handle
          Package: parallel.gpu

          Properties:
               ThreadBlockSize:  [10 1 1]
            MaxThreadsPerBlock:  1024
                      GridSize:  [1 1]
              SharedMemorySize:  0
                    EntryPoint:  '_Z13Simple_noCmexPdd'
            MaxNumLHSArguments:  1
               NumRHSArguments:  2
                 ArgumentTypes:  {'inout double vector' 'in double
                                   scalar'}

Methods, Events, Superclasses
```

在这个简单的 CUDA 核函数（Simple_noCmex.cu）中，核心句柄（handleCu）性质显示一个 inout 实参和一个 in 实参。最后一行为核函数的返回类型，始终应该为 void 类型，而输出值应该存回第一个或第二个输入实参中。

第 6 章　使用 CUDA 加速函数库

6.1　本章学习目标

c-mex 配置为运行 MATLAB 程序增加了很多可能性。通过调用现成的 CUDA 加速库，能做的甚至能远远超出 MATLAB 的极限。在 c-mex 中运用其他第三方函数库与普通 C/C++编程没有什么区别。

本章将学习以下内容：

- MATLAB 通过 c-mex 调用 CUDA 基本线性代数子程序（CUDA Basic Linear Algebra Subroutines，CUBLAS）。
- MATLAB 通过 c-mex 调用 CUDA 快速傅里叶变换库（CUDA FFT library，CUFFT）。
- 在基于标准模板库（Standard Template Library，STL）的 CUDA 中插入 C++模板库。

6.2　CUBLAS

CUBLAS 是 NVIDIA 提供的用于 GPU 加速的基本线性代数子程序（Basic Linear Algebra Subroutines，BLAS）库。它把 CUDA 过程进行了封装，因此我们可以在 c-mex 中较高的 API 级别上使用它，而无需接触 CUDA 的核心内容。本章将通过实例讲解在 c-mex 函数中调用 CUBLAS 库。

在用 CUBLAS 编写 c-mex 函数之前，首先我们了解一些有用信息。CUBLAS 与 NVIDIA CUDA 一起发布，在如下的 CUDA 安装路径中可以找到 CUBLAS 的函数库和头文件。注意，如果不是采用默认安装，系统中确切的安装路径可能与以下路径有所不同。与前面章节中列举的例子不同，这次没有调用 CUDA 编译器 nvcc，在函数中只调用了 CUDA 和 CUBLAS 运行时库。

- 编译时需要的头文件：

```
cublas_v2.h
cuda_runtime.h
```

- 链接时需要的函数库：

cublas.lib, cudart.lib	Windows
libculas.dylib, libcudart.dylib	Mac OS X
libculas.so, libcudart.so	Linux

这些文件一般在以下路径中：

Windows 64 位操作系统：

```
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\lib\x64\cublas.lib
C:\Program Fiels\NVIDIA GPU Computing Toolkit\CUDA\v5.0\include\cublas_v2.h
```

Windows 32 位操作系统：

```
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\lib\Win32\cublas.lib
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\include\cublas_v2.h
```

Mac OS X 操作系统：

```
/Developer/NVIDIA/CUDA-5.0/lib/libcublas.dylib
/Developer/NVIDIA/CUDA-5.0/include/cublas_v2.h
```

Linux 32 位操作系统：

```
/usr/local/cuda-5.0/lib/libcublas.so
/usr/local/cuda-5.0/include/cublas_v2.h
```

Linux 64 位操作系统：

```
/usr/local/cuda-5.0/lib64/libcublas.so
/usr/local/cuda-5.0/include/cublas_v2.h
```

在 c-mex 函数中会调用这些文件，因此在学习下一小节内容之前一定要找到并确认这些文件在系统中的位置。

6.2.1 CUBLAS 函数

与 MATLAB 一样，在 CUBLAS 函数库中，数据按列排列。这是一个很大的优势，尤其是在 c-mex 编程中，不需要考虑 c-mex 和我们感兴趣的库之间数据排列顺序的不同。同样，因为函数库提供了一系列重要且有用的线性代数函数，我们可以很容易地找出与原来 MATLAB 程序对应的部分。当在 MATLAB 程序中处理一个大数据集时，可以发现 CUBLAS 可广泛用于程序加速。

CUBLAS 函数可以分为四类：辅助函数、level-1 函数、level-2 函数和 level-3 函数。辅助函数主要提供处理 GPU 和内存资源包括数据副本的方法。调用这些函数时，不需要调用 CUDA 运行时库，表 6.1 列举了其中部分函数。表 6.2～表 6.4 列举了 level-1～level-3 的函数。

表 6.1　提供硬件资源控制的 CUBLAS 辅助函数

函 数 名	函 数 功 能
cublasCreate	初始化 CUBLAS 库，必须在所有 CUBLAS 库调用之前调用
cublasDestroy	释放 CUBLAS 库占用的硬件资源
cublasSetVector	把主机存储空间中的向量元素复制到 GPU 存储空间的向量中
cublasGetVector	把 GPU 存储空间中的向量元素复制到主机存储空间的向量中
cublasSetMatrix	把主机存储空间中的矩阵元素复制到 GPU 存储空间的矩阵中
cublasGetMatrix	把 GPU 存储空间中的矩阵元素复制到主机存储空间的矩阵中

表 6.2　进行基于标量和向量运算的 CUBLAS Level-1 函数

函 数 名	函 数 功 能
cublasIsamax	找到单精度向量中最大值元素的索引
cublasIdamax	找到双精度向量中最大值元素的索引
cublasSaxpy	向量与一个标量相乘，并将它加到另一个单精度向量中
cublasDdot	计算两个双精度向量的点积
cublasScrm2	计算单精度复数向量的欧几里得范数

表 6.3　进行矩阵和向量运算的 CUBLAS Level-2 函数

函 数 名	函 数 功 能
cublasSgemv	单精度矩阵和向量相乘
cublasDsbmv	双精度对称带状矩阵和向量相乘
cublasCsymv	单精度复数对称矩阵和向量相乘
cublasZhemv	双精度复数厄米特矩阵和向量相乘
cublasCher2	单精度复数厄米特秩-2 更新

表 6.4　进行矩阵和矩阵运算的 CUBLAS Level-3 函数

函 数 名	函 数 功 能
cublasSgemm	单精度矩阵乘法
cublasZgemm	双精度复数矩阵乘法
cublasDtrmm	双精度复数三角矩阵乘法
cublasChemm	单精度复数厄米特矩阵乘法
cublasCher2k	单精度复数厄米特秩 2k 更新

6.2.2　CUBLAS 矩阵乘法

本节通过例子讲解进行简单的矩阵与矩阵乘法的具体步骤，即

$$\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$$

其中，矩阵 $\boldsymbol{A}$ 的大小是 $M \times N$，矩阵 $\boldsymbol{B}$ 的大小是 $N \times P$，计算结果 $\boldsymbol{C}$ 的大小是 $M \times P$。

1．步骤 1

打开 MATLAB 命令窗口，创建一个新文件，将其保存为 cublasDemo.cpp。

2．步骤 2

在空文件 cublasDemo.cpp 中，输入以下代码：

```
#include "mex.h"

Void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
*prhs[])
{
}
```

正如你可能记住的那样，这是一个空的子例行程序，并从这开始 c-mex 程序。

3．步骤 3

接下来，检查输入数据类型是单精度还是浮点。然后，我们获得输入矩阵 ***A*** 和 ***B*** 的指针以及它们的大小。为简单起见，假设输入数据类型是单精度，否则退出编程：

```
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
*prhs[])

 {
      if (nrhs != 2)
          mexErrMsgTxt("Invaid number of input arguments");

      if (!mxIsSingle(prhs[0]) && !mxIsSingle(prhs[1]))
          mexErrMsgTxt("input matrices must be single ");

       float* A = (float*)mxGetData(prhs[0]);
       float* B = (float*)mxGetData(prhs[1]);

       int numARows = mxGetM(prhs[0]);
       int numACols = mxGetN(prhs[0]);
       int numBRows = mxGetM(prhs[1]);
       int numBCols = mxGetN(prhs[1]);

       If (numACols != numBRows)
          mexErrMsgTxt("Invalid matrix dimension");
}
```

4．步骤 4

生成输出矩阵 ***C***，它的大小由矩阵 ***A*** 的行数和矩阵 ***B*** 的列数决定：

```
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
*prhs[])
{
      if (nrhs != 2)
          mexErrMsgTxt("Invaid number of input arguments");

      if (!mxIsSingle(prhs[0]) && !mxIsSingle(prhs[1]))
          mexErrMsgTxt("input matrices must be single ");

      float* A = (float*)mxGetData(prhs[0]);
      float* B = (float*)mxGetData(prhs[1]);

      int numARows = mxGetM(prhs[0]);
      int numACols = mxGetN(prhs[0]);
      int numBRows = mxGetM(prhs[1]);
      int numBCols = mxGetN(prhs[1]);
      int numCRows = numARows;
      int numCCols = numBCols;

      plhs[0] = mxCreateNumericMatrix(numCRows, numCCols, mxSINGLE_
      CLASS, mxREAL);
      float* C = (float*)mxGetData(plhs[0]);
}
```

5．步骤 5

现在在 GPU 设备上创建用于存储矩阵数据的存储空间。分别用 cudaMalloc 和 cudaFree 函数来分配和释放内存。这些函数在 cuda_runtime.h 中定义，所以在程序最开始的地方要声明这个头文件：

```
#include "mex.h"
#include <cuda_runtime.h>

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
*prhs[])
{

      if (nrhs != 2)
          mexErrMsgTxt("Invaid number of input arguments");

      if (!mxIsSingle(prhs[0]) && !mxIsSingle(prhs[1]))
          mexErrMsgTxt("input matrices must be single ");

      float* A = (float*)mxGetData(prhs[0]);
      float* B = (float*)mxGetData(prhs[1]);
```

```
        int numARows = mxGetM(prhs[0]);
        int numACols = mxGetN(prhs[0]);
        int numBRows = mxGetM(prhs[1]);
        int numBCols = mxGetN(prhs[1]);
        int numCRows = numARows;
        int numCCols = numBCols;

        plhs[0] = mxCreateNumericMatrix(numCRows, numCCols, mxSINGLE_
        CLASS, mxREAL);
        float* C = (float*)mxGetData(plhs[0]);

        float *deviceA, *deviceB, *deviceC;
        cudaMalloc(&deviceA, sizeof(float) * numARows * numACols);
        cudaMalloc(&deviceB, sizeof(float) * numBRows * numBCols);
        cudaMalloc(&deviceC, sizeof(float) * numCRows * numCCols);

        // insert cuBLAS function(s) here

        cudaFree(deviceA);
        cudaFree(deviceB);
        cudaFree(deviceC);
    }
```

可以注意到，我们只是分配和释放了 GPU 中的内存，把 CUBLAS 代码插入其中。

6．步骤 6

现在开始添加 CUBLAS 代码。首先在程序开始部分添加另一个头文件 cublas_v2.h：

```
#include "mex.h"
#include <cuda_runtime.h>
#include <cublas_v2.h>

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
*prhs[])
{
        if (nrhs != 2)
            mexErrMsgTxt("Invaid number of input arguments");

        if (!mxIsSingle(prhs[0]) && !mxIsSingle(prhs[1]))
            mexErrMsgTxt("input matrices must be single ");

        float* A = (float*)mxGetData(prhs[0]);
        float* B = (float*)mxGetData(prhs[1]);
```

```
    int numARows = mxGetM(prhs[0]);
    int numACols = mxGetN(prhs[0]);
    int numBRows = mxGetM(prhs[1]);
    int numBCols = mxGetN(prhs[1]);
    int numCRows = numARows;
    int numCCols = numBCols;

    plhs[0] = mxCreateNumericMatrix(numCRows, numCCols, mxSINGLE_
    CLASS, mxREAL);
    float* C = (float*)mxGetData(plhs[0]);

    float *deviceA, *deviceB, *deviceC;
    cudaMalloc(&deviceA, sizeof(float) * numARows * numACols);
    cudaMalloc(&deviceB, sizeof(float) * numBRows * numBCols);
    cudaMalloc(&deviceC, sizeof(float) * numCRows * numCCols);

    cublasHandle_t handle;
    cublasCreate(&handle);
    cublasSetMatrix(numARows,
                    numACols,
                    sizeof(float),
                    A,
                    numARows,
                   deviceA,
                   numARows);
   cublasSetMatrix(numBRows,
                   numBCols,
                   sizeof(float),
                   B,
                   numBRows,
                   deviceB,
                   numBRows);

   cublasDestroy(handle);
   cudaFree(deviceA);
   cudaFree(deviceB);
   cudaFree(deviceC);
}
```

头文件 cublas_v2.h 中包含了 CUBLAS 库的函数原型。首先创建 CUBLAS 句柄，最后通过调用 cublasDestroy(…) 删除句柄，这是我们调用所有 CUBLAS 函数之前要做的事情。然后，通过调用 cublasSetMatrix(…) 来准备矩阵，这样可以将矩阵从主机存储空间复制到分配好的 GPU 设备存储空间中去。注意，将数据移动到 GPU 设备时，我们并没有调用 cudaMemcoy(…)，这里因为文件 cublasSetMatrix(…) 已经在后台完成了这一工作。

7．步骤 7

本步骤仅仅调用其中矩阵和矩阵相乘函数 PCublasSgemm(…)，这个函数在 GPU 上进行实际的运算。正如它的函数名一样，这个函数的作用是实现单精度数据类型的矩阵相乘。

```
#include "mex.h"
#include <cuda_runtime.h>
#include <cublas_v2.h>

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
*prhs[])
{
      if (nrhs != 2)
          mexErrMsgTxt("Invaid number of input arguments");

      if (!mxIsSingle(prhs[0]) && !mxIsSingle(prhs[1]))
          mexErrMsgTxt("input matrices must be single");

      float* A = (float*)mxGetData(prhs[0]);
      float* B = (float*)mxGetData(prhs[1]);
int numARows = mxGetM(prhs[0]);
int numACols = mxGetN(prhs[0]);
int numBRows = mxGetM(prhs[1]);
int numBCols = mxGetN(prhs[1]);
int numCRows = numARows;
int numCCols = numBCols;

plhs[0] = mxCreateNumericMatrix(numCRows, numCCols, mxSINGLE_
CLASS, mxREAL);
float* C = (float*)mxGetData(plhs[0]);

float *deviceA, *deviceB, *deviceC;
cudaMalloc(&deviceA, sizeof(float) * numARows * numACols);
cudaMalloc(&deviceB, sizeof(float) * numBRows * numBCols);
cudaMalloc(&deviceC, sizeof(float) * numCRows * numCCols);

cublasHandle_t handle;
cublasCreate(&handle);
cublasSetMatrix(numARows,
                  numACols,
                  sizeof(float),
                  A,
                  numARows,
                  deviceA,
                  numARows);
```

```
cublasSetMatrix(numBRows,
                numBCols,
                sizeof(float),
                B,
                numBRows,
                deviceB,
                numBRows);

float alpha = 1.0f;
float beta = 0.0f;
cublasSgemm(handle,
            CUBLAS_OP_N,
            CUBLAS_OP_N,
            numARows,
            numBCols,
            numACols,
            &alpha,
            deviceA,
            numARows,
            deviceB,
            numBRows,
            &beta,
            deviceC,
                numCRows);

    cublasGetMatrix(numCRows,
                    numCCols,
                    sizeof(float),
                    deviceC,
                    numCRows,
                    C,
                    numCRows);

    cublasDestroy(handle);
    cudaFree(deviceA);
    cudaFree(deviceB);
    cudaFree(deviceC);
}
```

文件 cublasSgemm(…) 完成所有低层次 CUDA 工作，并且将结果返回到为矩阵 ***C*** 分配的 GPU 内存中，然后文件 cublasGetMatrix(…) 将结果从 GPU 存储空间复制到主机存储中。我们不需要设置数据的线程块和线程的大小，CUBLAS 会自动设定它们的维度，执行内核函数，并返回输出结果。

8．步骤 8

现在 c-mex 编码完成，进入 MATLAB 命令窗口，输入 mex 命令来实际运行程序：

```
mex cublasDemo.cpp -lcudart -lcublas -L"C:\Program Files\NVIDIA GPU
Computing Toolkit\CUDA\v5.0\lib\x64" -v -I"C:\Program Files\NVIDIA GPU
Computing Toolkit\CUDA\v5.0\include"
```

其中，各选项含义如下：

- -lcudart：表明在调用 CUDA 运行时库。具体而言，我们在调用两个基本 CUDA 函数 cudaMalloc(…)和 cudaFree(…)。
- -lcublas：表明在调用 CUBLAS 库。具体而言，我们在调用 cublasXXX 函数。
- -L*dir*：dir 是 CUDA 和 CUBLAS 库所在的目录。
- -I*dir*：dir 是 CUDA 和 CUBLAS 头文件所在的目录。

9．步骤 9

在 Windows 64 位操作系统中，运行步骤 8 中的 MATLAB 命令会生成 c-mex 文件 cublasDemo1.mexw64，在 MATLAB 命令窗口中调用生成的函数来进行乘法计算：

```
>> A = single(rand(200, 300));
>> B = single(rand(300, 400));
>> C = cublasDemo(A, B);
```

可以用 cublasExample.m 在示例代码目录中测试这些代码。

6.2.3 使用 Visual Profiler 进行 CUBLAS 分析

让我们使用 NVIDIA Visual Profiler 对 CUBLAS 函数 cublasSgemm(…) 做更深入的了解。NVIDIA Visual Profiler 除了分析运行时间外，还可以了解程序运行时的具体细节。

首先，打开 NVIDIA Visual Profiler，然后如第 3 章的介绍，通过分析器打开 MATLAB，当 MATLAB 窗口打开后，将我们编译的 c-mex 文件所处位置设置为当前工作路径，然后如图 6.1 所示运行 c-mex 函数。

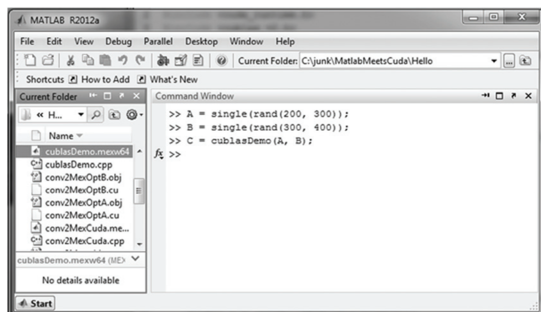


图 6.1 在 MATLAB 中运行 cublasDemo

运行 c-mex 函数后，关闭 MATLAB 窗口退出程序的执行。MATLAB 关闭后，返回 NVIDIA Visual Profiler，这时 NVIDIA Visual Profiler 会停止对 MATLAB 的信息收集并在窗口中生成结果，如图 6.2 所示。

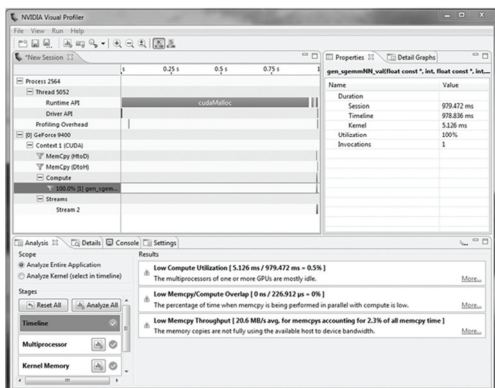


图 6.2　用 NVIDIA Visual Profiler 运行 cublasDemo

将 CUBLAS 函数调用的地方展开，展开后除了时间分析外，还给出了其他细节，如图 6.3 所示。

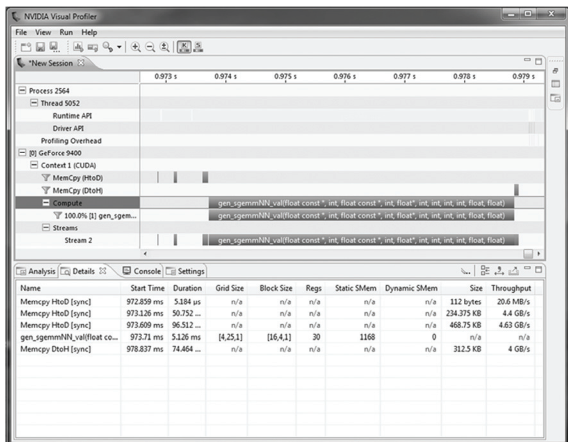
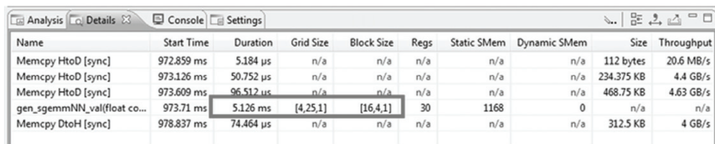


图 6.3　Visual Profiler 运行 cublasDemo 展开 GPU 运算细节

在这个例子中，可以看到 CUDA 函数调用的实际时间线轴。可以看到，对于矩阵和矩阵的计算，调用了函数 gen_sgemmNN_val(. . .)，如图 6.4 所示。



Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput
Memcpy HtoD [sync]	972.859 ms	5.184 μs	n/a	n/a	n/a	n/a	n/a	112 bytes	20.6 MB/s
Memcpy HtoD [sync]	973.126 ms	50.752 μs	n/a	n/a	n/a	n/a	n/a	234.375 KB	4.4 GB/s
Memcpy HtoD [sync]	973.609 ms	96.512 μs	n/a	n/a	n/a	n/a	n/a	468.75 KB	4.63 GB/s
gen_sgemmNN_val(float co...	973.71 ms	5.126 ms	[4,25,1]	[16,4,1]	30	1168	0	n/a	n/a
Memcpy DtoH [sync]	978.837 ms	74.464 μs	n/a	n/a	n/a	n/a	n/a	312.5 KB	4 GB/s

图 6.4　Visual Profiler 运行 cublasDemo 后 cublas 函数细节

观察 Details 选项栏，可以看到每个 CUDA 函数调用所用的具体时间，特别地，可以得到内核函数中更多关于线程网格和线程块大小的详细信息。CUBLAS 自动设定了线程网格和线程块的大小。此例中，

线程网格大小：[4, 25, 1]

线程块大小：[16, 4, 1]

由此我们可以知道，CUBLAS 为大小为 200×400 的结果矩阵 **C** 共分配了 6400（16×4×4×25）个线程。

综上所述，c-mex 函数包含两个头文件（见图 6.5）。当在 MATLAB 中编译 mex 代码时，同时指出了所调用的库以及库的位置。通过这种方式，扩展 c-mex 函数包含 CUBLAS 库。利用 CUDA 运行时库来为输入和输出数据分配和释放内存空间。CUBLAS 函数除了执行矩阵和矩阵乘法的核心运算外，也负责数据在主机和 GPU 设备之间转移。

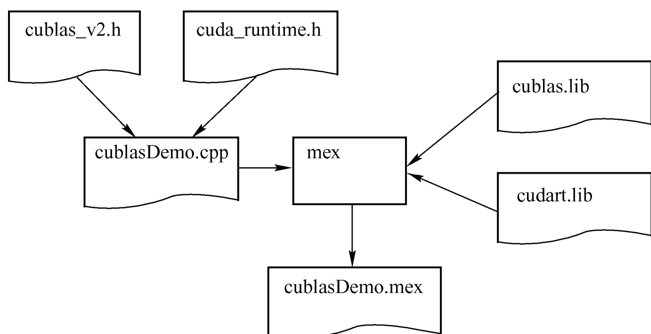


图 6.5　cublasDemo c-mex 生成过程

6.3　CUFFT

CUFFT 是 NVIDIA 提供的具有 GPU 加速的快速傅里叶变换（FFT）函数库。与 CUBLAS 一样，CUFFT 隐藏了 CUDA 的细节，所以可以使用简单的界面来利用 NVIDIA GPU 的计算能力。该函数库支持兼容 FFTW㊀的数据布局，所以可以无缝整合现有的 FFTW 代码。

和 CUBLAS 一样，CUFFT 也需要 C 头文件和库，这些文件位于 CUBLAS 头文件和库文件所在目录。

编译所需的头文件：

㊀FFTW（The Faster Fouriev Transform in the west）是一个快速计算 FFT 的标准 C 语言程序集。

```
cufft.h
cuda_runtime.h
```

链接所需的库：

cufft.lib，cudart.lib	Windows
libcufft.dylib，libcudart.dylib	Mac OS X
libcufft.so，libcudart.so	Linux

64 位 Windows 操作系统：

```
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\lib\x64\cufft.lib
C:\Program Fiels\NVIDIA GPU Computing Toolkit\CUDA\v5.0\include\cufft.h
```

32 位 Windows 操作系统：

```
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\lib\Win32\cufft.lib
C:\Program Files\NVIDIA GPU ComputingToolkit\CUDA\v5.0\include\cufft.h
```

Mac OS X 操作系统：

```
/Developer/NVIDIA/CUDA-5.0/lib/libcufft.dylib
/Developer/NVIDIA/CUDA-5.0/include/cufft.h
```

32 位 Linux 操作系统：

```
/usr/local/cuda-5.0/lib/libcufft.so
/usr/local/cuda-5.0/include/cufft.h
```

64 位 Linux 操作系统：

```
/usr/local/cuda-5.0/lib64/libcufft.so
/usr/local/cuda-5.0/include/cufft.h
```

6.3.1　通过 CUFFT 进行二维 FFT 运算

下面再次用一个简单的例子来介绍在 c-mex 函数中怎样调用 CUFFT 函数库。本例中，将随机生成一个二维实矩阵，进行二维 FFT 运算，并将其复数结果返回 MATLAB。

1．步骤 1

打开 MATLAB 命令窗口，创建一个新文件，将其保存为 cufftDemo.cpp。

2．步骤 2

在空文件 cuftDemo.cpp 中，输入以下代码：

```
#include "mex.h"

Void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
*prhs[])
{
}
```

与 CUBLAS 例子中相同，这是空子例行程序。

3．步骤 3

在这步中，检查输入数据类型是单精度还是浮点型：

```
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
*prhs[])
{
    if (nrhs != 1)
        mexErrMsgTxt("Invaid number of input arguments");

    if (!mxIsSingle(prhs[0]) && !mxIsSingle(prhs[1]))
        mexErrMsgTxt("input data type must be single");

    float* A = (float*)mxGetData(prhs[0]);

    int numARows = mxGetM(prhs[0]);
    int numACols = mxGetN(prhs[0]);

    float *deviceA;

    cudaMalloc(&deviceA, sizeof(float) * numARows * numACols);
    cudaMemcpy(deviceA, A, numARows * numACols * sizeof(float),
             cudaMemcpyHostToDevice);
}
```

4．步骤 4

这步中，我们在设备中为输入矩阵 ***A*** 创建存储空间，并将数据复制到 GPU 设备中：

```
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
*prhs[])
{
    if (nrhs != 1)
        mexErrMsgTxt("Invaid number of input arguments");

    if (!mxIsSingle(prhs[0]) && !mxIsSingle(prhs[1]))
        mexErrMsgTxt("input data type must be single");

    float* A = (float*)mxGetData(prhs[0]);

    int numARows = mxGetM(prhs[0]);
    int numACols = mxGetN(prhs[0]);

    float *deviceA;
```

```
    cudaMalloc(&deviceA, sizeof(float) * numARows * numACols);
    cudaMemcpy(deviceA, A, numARows * numACols * sizeof(float),
               cudaMemcpyHostToDevice);
}
```

5. 步骤 5

现在，在 GPU 中创建空间存储 FFT 输出的复数据：

```
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
*prhs[])
{
    if (nrhs != 1)
        mexErrMsgTxt("Invaid number of input arguments");

    if (!mxIsSingle(prhs[0]) && !mxIsSingle(prhs[1]))
        mexErrMsgTxt("input data type must be single");

    float* A = (float*)mxGetData(prhs[0]);

    int numARows = mxGetM(prhs[0]);
    int numACols = mxGetN(prhs[0]);

    float* deviceA;

    cudaMalloc(&deviceA, sizeof(float) * numARows * numACols);
    cudaMemcpy(deviceA, A, numARows * numACols * sizeof(float),
               cudaMemcpyHostToDevice);

    int outRows = numARows /2 + 1;
    int outCols = numACols;
    cufftComplex* deviceOut;
    cudaMalloc(&deviceOut, sizeof(cufftComplex) * outRows * outCols);

}
```

这里我们必须要格外注意，MATLAB 数据是按列存储，这表示同一列中的数据在内存空间中是连续存放的。然而，CUFFT 数据是按行存储的。CUFFT 希望数据是按行连续存储，而不是像 MATLAB 那样按列存储。

表 6.5 是 CUFFT API 参考中给出的输入输出数据大小汇总。这里的 2D 和 R2C（实数-复数）分别是我们关注的维度和类型。N_2 是按列变换的数据，当在内存空间中连续移动时，N_2 快速变化。所以在 MATLAB 数据类型中，N_1 是行维度，N_2 是列维度，MATLAB 中的列维度和行维度在 CUFFT 中分别对应 N_1 和 N_2。如果我们不注意维度和数据布局，得出的结果将发生转置。

表 6.5　输入和输出数据大小（来自 CUFFT API 参考）

维　度	FFT 类型	输入数据大小	输出数据大小
1D	C2C（复数-复数）	N_1 cufftComplex	N_1 cufftComplex
	C2R（复数-实数）	$[N_1/2]+1$ cufftComplex	N_1 cufftReal
	R2C（实数-复数）	N_1 cufftReal	$[N_1/2]+1$ cufftComplex
2D	C2C（复数-复数）	N_1 N_2 cufftComolex	N_1N_2 cufftComplex
	C2R（复数-实数）	$N_1([N_1/2]+1)$ cufftComplex	N_1N_2 cufftReal
	R2C（实数-复数）	N_1N_2 cufftReal	$N_1([N_2/2]+1)$ cufftComplex
3D	C2C（复数-复数）	$N_1N_2N_3$ cufftComplex	$N_1N_2N_3$ cufftComplex
	C2R（复数-实数）	N_1N_2 $([N_3/2]+1)$ cufftComplex	N_1 N_2 N_3 cufftReal
	R2C（实数-复数）	$N_1N_2N_3$ cufftReal	$N_1N_2([\ N_3\ /2+1])$ cufftComplex

- MATLAB 中的行维度 M 对应 CUFFT 中的 N_2。
- MATLAB 中的列维度 N 对应 CUFFT 中的 N_1。

所以，输出的行数是实际行数的一半加 1。看一下函数参考中给出的函数 cufftPlan2d(…)的定义，这个函数将在下一步中调用。

cufftResult cufftPlan2d(cufftHandle *plan, int nx, int ny, cufftType type) 是根据指定的信号大小和数据类型创建 2D FFT 配置。例如，输入信息如下：

plan	指向 cufftHandle 对象的指针
nx	x 维度 FFT 变换的大小（行数）
ny	y 维度 FFT 变换大小（列数）
type	FFT 变换数据类型（例如，对于单精度数据类型，CUFFT_C2R 将复数变为实数）

对于输入参数 nx 和 ny，我们分别传递 numACols 和 numARows 的值。

6．步骤 6

输入和输出数据矩阵准备就绪后，调用 CUFFT 函数。像 CUBLAS 例子中那样，首先创建它的句柄：

```
#include "mex.h"
#include <cuda_runtime.h>
#include <cufft.h>

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, mxArray *prhs[])
{
     if (nrhs != 1)
         mexErrMsgTxt("Invaid number of input arguments");

if (!mxIsSingle(prhs[0]) && !mxIsSingle(prhs[1]))
    mexErrMsgTxt("input data type must be single");
```

```
    float* A = (float*)mxGetData(prhs[0]);

    int numARows = mxGetM(prhs[0]);
    int numACols = mxGetN(prhs[0]);

    float *deviceA;

    cudaMalloc(&deviceA, sizeof(float) * numARows * numACols);
    cudaMemcpy(deviceA, A, numARows * numACols * sizeof(float),
               cudaMemcpyHostToDevice);

    int outRows = numARows /2 + 1;
    int outCols = numACols;
    cufftComplex* deviceOut;
    cudaMalloc(&deviceOut, sizeof(cufftComplex) * outRows * outCols);

    cufftHandle plan;
    cufftPlan2d(&plan, numACols, numARow, CUFFT_R2C);
    cufftExecR2C(plan, deviceA, deviceOut);

    cufftDestroy(plan);
    cudaFree(deviceA);
}
```

在 cufftPlan2d(…) 中，我们确定了输入矩阵的大小以及希望进行何种运算。一旦制定了这一计划，在之后的程序中如果需要的话可以再次利用，不需要时可以调用 cufftDestroy(…) 函数将其清除。实际的计算在 cufftExecR2C(…)函数中进行，在输出数组 deviceOut 中可得到输出。

7．步骤 7

在 GPU 内存中可以得到 FFT 计算数据。将这些计算数据返回到主机存储器中，之后就可以在 MATLAB 中使用它了。

```
#include "mex.h"
#include <cuda_runtime.h>
#include <cufft.h>

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, mxArray *prhs[])
{
    if (nrhs != 1)
        mexErrMsgTxt("Invaid number of input arguments");

    if (!mxIsSingle(prhs[0]) && !mxIsSingle(prhs[1]))
        mexErrMsgTxt("input data type must be single");

    float* A = (float*)mxGetData(prhs[0]);

    int numARows = mxGetM(prhs[0]);
```

```
    int numACols = mxGetN(prhs[0]);

    float *deviceA;

    cudaMalloc(&deviceA, sizeof(float) * numARows * numACols);
    cudaMemcpy(deviceA, A, numARows * numACols * sizeof(float),
               cudaMemcpyHostToDevice);

    int outRows = numARows /2 + 1;
    int outCols = numACols;
    cufftComplex* deviceOut;
    cudaMalloc(&deviceOut, sizeof(cufftComplex) * outRows * outCols);

    cufftHandle plan;
    cufftPlan2d(&plan, numACols, numARows, CUFFT_R2C);
    cufftExecR2C(plan, deviceA, deviceOut);

    float* out = (float*)mxMalloc(sizeof(cufftComplex) * outRows *
                  outCols);
    cudaMemcpy(out, deviceOut, outRows * outCols * sizeof(cufftComplex),
              cudaMemcpyDeviceToHost);

    plhs[0] = mxCreateNumericMatrix(outRows, outCols, mxSINGLE_CLASS,
                                   mxCOMPLEX);
    float* real = (float*)mxGetPr(plhs[0]);
    float* imag = (float*)mxGetPi(plhs[0]);
    float* complex = out;
    for (int c = 0; c < outCols; ++c)
    {
        for (int r = 0; r < outRows; ++r)
        {
            *real++ = *complex++;
            *imag++ = *complex++;
        }
    }

    mxFree(out);
    cufftDestroy(plan);
    cudaFree(deviceA);
}
```

与 CUBLAS 不同，我们用 CUDA 函数分配和释放内存，且在 GPU 和 c-mex 之间相互传送数据。但除了这三个 CUDA 函数外，不需要定义我们自己的内核函数，也不需要确定线程块和线程的大小。正如第 4 章中讨论的那样，MATLAB 把复数存放在两个独立的矩阵中：一个存储实部，一个存储虚部。CUFFT 将复数一个一个存储在一块内存空间中。所以在这步中，我们要为 MATLAB 创建一个复数

矩阵，并把实部和虚部数据复制到相应的矩阵中。

8．步骤 8

现在我们已经实现了 c-mex 代码。进入 MATLAB 命令窗口，然后编译我们的代码：

```
mex cufftDemo.cpp -lcudart -lcufft -L"C:\Program Files\NVIDIA GPU Computing
Toolkit\CUDA\v5.0\lib\x64" -v -I"C:\Program Files\NVIDIA GPU Computing
Toolkit\CUDA\v5.0\include"
```

各选项含义如下：

- -lcudart：表明在调用 CUDA 运行时库。具体而言，我们调用了两个基本 CUDA 函数：cudaMalloc(…)和 cudaFree(…)。
- -lcufft：表明在调用 CUFFT 库。具体而言，我们调用了 cufftXXX 函数调用。
- -L*dir*：dir 是 CUDA 和 CUFFT 库所在的目录。
- -I*dir*：dir 是 CUDA 和 CUFFT 头文件所在的目录。

9．步骤 9

如在 64 位 Windows 操作系统下进行编译，前面 c-mex 代码编译会产生 c-mex 文件 cublasDemo.mexw64。在 MATLAB 命令窗口中调用我们的函数来进行以下乘法：

```
>> A = single(rand(4, 4));
>> B = fft2(A)

B =
    9.7001              0.5174 - 0.9100i   1.6219              0.5174 + 0.9100i
   -0.6788 + 0.4372i   0.7316 + 0.5521i  -0.1909 - 1.0807i  -1.3437 + 0.2664i
    1.3043             -0.9332 + 0.6233i  -2.0830             -0.9332 - 0.6233i
   -0.6788 - 0.4372i  -1.3437 - 0.2664i  -0.1909 + 1.0807i   0.7316 - 0.5521i

>> C = cufftDemo(A)

C =
    9.7001 - 0.0000i   0.5174 - 0.9100i   1.6219 - 0.0000i   0.5174 + 0.9100i
   -0.6788 + 0.4372i   0.7316 + 0.5521i  -0.1909 - 1.0807i  -1.3437 + 0.2664i
    1.3043 + 0.0000i  -0.9332 + 0.6233i  -2.0830 + 0.0000i  -0.9332 - 0.6233i
```

比较两个结果：一个用 MATLAB fft2 函数，另一个用 CUFFT c-mex 函数。注意到 cufftDemo(A) 返回结果比 MATLAB fft2 函数少一行，这一输出结果依据的是其 API 规范（见表 6.5）。

FFT 类型	输入数据大小	输出数据大小
R2C（实数-复数）	**N_1N_2 cufftReal**	**N_1([N_2/2]+1) cufftComplex**

本例中，N_1=4，N_2=4，这样最终输出结果为三行。

在这个例子中，缺少的这行可以用第二行的复共轭计算得到，如下所示：

```
>> D = [C; conj([flipud([C(2:2,1), fliplr(C(2:2, 2:4))])])]

D =
   9.7001 - 0.0000i   0.5174 - 0.9100i   1.6219 - 0.0000i   0.5174 + 0.9100i
  -0.6788 + 0.4372i   0.7316 + 0.5521i  -0.1909 - 1.0807i  -1.3437 + 0.2664i
   1.3043 + 0.0000i  -0.9332 + 0.6233i  -2.0830 + 0.0000i  -0.9332 - 0.6233i
  -0.6788 - 0.4372i  -1.3437 - 0.2664i  -0.1909 + 1.0807i   0.7316 - 0.5521i
```

6.3.2 用 Visual Profiler 进行 CUFFT 时间分析

采用 CUBLAS 的例子，采用 NVIDIA Visual Profiler 对 cufft 函数进行更深的认识。

打开 NVIDIA Visual Profiler，创建一个新的对话。在 MATLAB 窗口中，设置编译后的 c-mex 文件的位置为当前路径，然后运行 c-mex 函数，如图 6.6 所示。

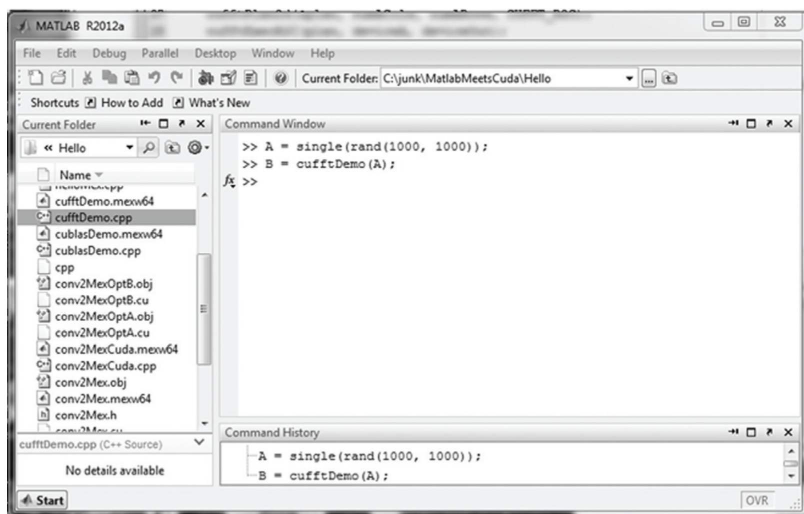


图 6.6 cufftDemo 在 MATLAB 中的运行

运行 c-mex 函数后，关闭 MATLAB 窗口退出 MATLAB。MATLAB 关闭后，返回 NVIDIA Visual Profiler。此时 NVIDIA Visual Profiler 完成了对 MATLAB 信息的搜集，并且在窗口中生成了分析结果。如果 NVIDIA Visual Profiler 弹出如图 6.7 所示窗口，则临时在 c-mex 函数末尾处添加 cudaDeviceRest( ) 函数，以确保刷新 CUDA 配置文件。

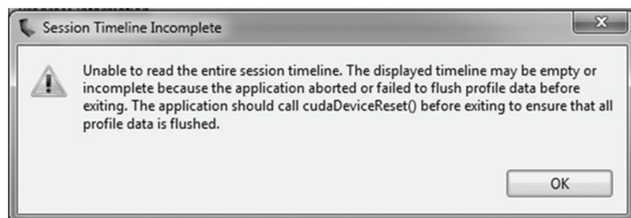


图 6.7 NVIDIA Visual Profiler 会话时间轴不完整消息

```
    mxFree(out);
    cufftDestroy(plan);
    cudaFree(deviceA);
    cudaFree(deviceB);

    cudaDeviceReset();
}
```

在 MATLAB 关闭后，让我们看看 CUFFT 执行情况的分析细节，如图 6.8 所示。

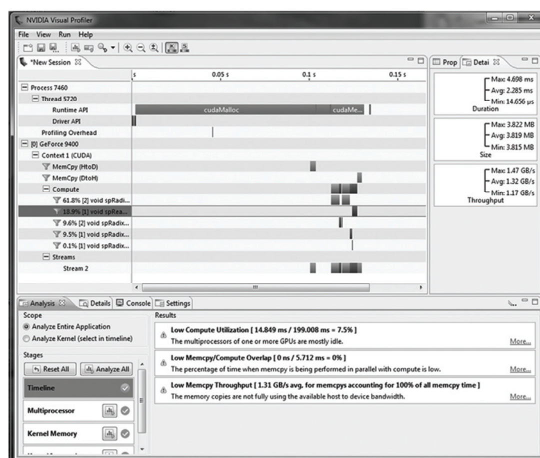


图 6.8　Visual Profiler 分析 cufftDem

在 cufft 函数调用的地方展开，展开后，屏幕显示除时间分析外的其他细节。我们可以更好地了解定义的 cufft 函数的内部情况。注意这一函数创建了多个不同线程大小的内核函数，如图 6.9 所示。

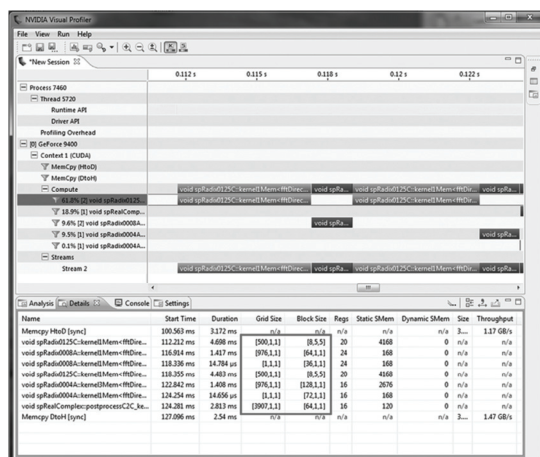


图 6.9　GPU 运算中 Visual Profiler 的 cufftDemo 分析细节

综上所述，如 CUBLAS 实例所示，c-mex 函数共包含两个头文件。在 MATLAB 中用 mex 编译代码时，我们指定所采用的函数库以及函数库的位置。所采用的 cuda 运行时函数用来分配和释放内存，以及在主机和设备间传递输入输出数据。在调用 cufft 函数时，必须考虑 MATLAB 的数据布局是否与 CUFFT 兼容。CUFFT 要求按行输入数据，但 MATLAB 则是按列顺序。还需要格外注意的是，MATLAB 中的行和列是如何映射到 CUFFT 中的 nx 和 ny 中的，如图 6.10 所示。

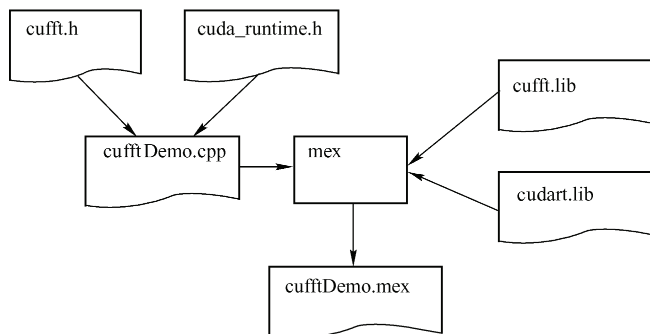


图 6.10　cufftDemo c-mex 创建过程

6.4　Thrust

Thrust 是一个 C++模板库，它使得用户能够使用 CUDA 实现高性能并行应用。Thrust 提供了丰富的数据并行基元，例如扫描、排序、归约。可以利用这些在 Thrust 中实现的算法来写出更简洁、更具可读性的程序。因为 Thrust 是模板库，所以不需要链接其他任何库，只需要把它的头文件包含在 c-mex 函数中。

如果使用 CUDA 4.0 或更高版本，Thrust 应该已经安装在系统中了。需要被包含的文件位于头文件目录。

- Windows 32 位和 64 位操作系统：

C:\Program Fiels\NVIDIA GPU Computing Toolkit\CUDA\v5.0\include

- Mac OS X 操作系统:

/Developer/NVIDIA/CUDA-5.0/include

- Linux 32 位和 64 位操作系统：

/usr/local/cuda-5.0/include

6.4.1　通过 Thrust 排序

1．步骤 1

打开 MATLAB 命令窗口，创建一个新文件，并保存为 thrustDemo.cpp。

2．步骤 2

在 thrust.cpp 空文件中，输入以下代码：

```
#include "mex.h"

Void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
*prhs[])
{
}
```

这是一个空的子例行程序。

3．步骤 3

在本步骤中，检查输入数据是单精度还是浮点型，并且准备浮点型的输出数据。然后调用 getSum(…)，在这个函数中调用 thrust 函数。下面例子中的第三行是告诉编译器将要使用 getSum(…) 函数，这个函数在这个文件之外已经完成了：

```
#include "mex.h"

extern float getSum(float* A, int size);
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
*prhs[])
{
    if (nrhs != 1)
        mexErrMsgTxt("Invaid number of input arguments");

    if (!mxIsSingle(prhs[0]) && !mxIsSingle(prhs[1]))
        mexErrMsgTxt("input data type must be single");

    float* A = (float*)mxGetData(prhs[0]);
    int numARows = mxGetM(prhs[0]);
    int numACols = mxGetN(prhs[0]);
    int numElem = (numARows > numACols) ? numARows: numACols;

    plhs[0]=mxCreateNumericMatrix(1, 1, mxSINGLE_CLASS, mxCOMPLEX);
    float* B = (float*)mxGetData(plhs[0]);

    *B = getSum(A, numElem);
}
```

4．步骤 4

现在创建 thrustSum.cu 文件，在这个文件中所有的 thrust 函数将会被编译。采用 nvcc 编译器编译 thrust 代码，nvcc 编译器将在后台为 CUDA 生成二进制文件，创建 thrustSum.cu 并输入以下代码：

```
#include <thrust/reduce.h>
#include <thrust/device_vector.h>

float getSum(float* A, int size)
{
    thrust::device_vector<float> deviceA(A, A + size)
    return thrust::reduce(deviceA.begin(),
                          deviceA.end(),
                          (float)0.0f,
                          thrust::plus<float>());
}
```

5．步骤 5

编译并生成目标文件 thrustSum.obj，c-mex 主函数会链接这个文件：

```
>> system('nvcc -c thrustSum.cu');
```

6．步骤 6

编译 c-mex 主函数并链接 thrustSum.obj：

```
>> mex thrustDemo.cpp thrustSum.obj -lcudart -L"C:\Program Files\NVIDIA
GPU Computing Toolkit\CUDA\v5.0\lib\x64" -v -I"C:\Program Files\NVIDIA
GPU Computing Toolkit\CUDA\v5.0\include"
```

7．步骤 7

现在生成了在 MATLAB 中可以调用的 c-mex 函数，在 MATLAB 命令窗口中可以运行此函数：

```
>> thrustDemo(single([1 2 3 4]))

ans =
    10

>> thrustDemo(single(rand(1, 10000)))

ans =
    4.9986e+03
```

6.4.2 采用 Visual Profiler 分析 Thrust

c-mex 函数调用 Thrust 库时，可以同样使用 NVIDIA Visual Profiler 查看细节，包括执行哪种内核函数以及程序如何执行。像在 CUBLAS 和 CUFFT 中操作的那样，用 MATLAB 运行 NVIDIA Visual Profiler，之后在 MATLAB 命令窗口中输入以下代码运行：

```
>> thrustDemo(single(rand(1, 12345)))

ans =
    6.1485e+03
```

如果 MATLAB 停止运行，且 NVIDIA Visual Profiler 显示需要 cudaDeviceRest( ) 函数的错误，可以把这个函数包含在 c-mex 函数的最后，如下所示：

```
    *B = getSum(A, numElem);

    cudaDeviceReset();
}
```

两个内核函数在归约算法中被定义。每个内核函数都有一个一维的线程网格和线程块大小，如图 6.11 所示。

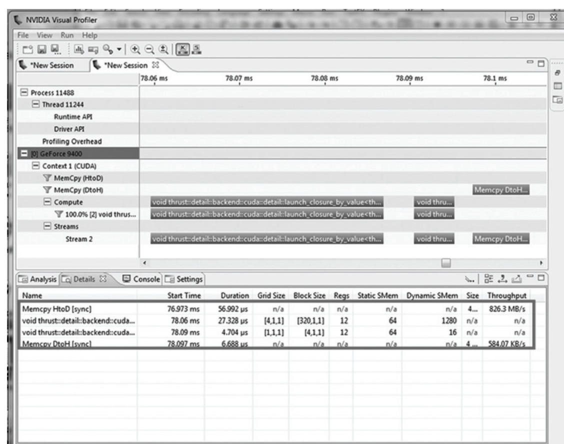


图 6.11　Visual Profiler 分析 thrustDemo

Thrust 是一个 C++模板库，在调用它时（见图 6.12），需要进行两步编译，首先用 nvcc 编译，然后再用 mex 编译。创建 thrustSum.cu 文件，用于存储所有 Thrust 相关的内容。然后，用 nvcc 编译器编译并生成目标文件。首先必须声明包含这个函数或其他感兴趣函数的头文件。然后，编译 c-mex 主函数并用目标文件链接，生成最终的 c-mex 文件。

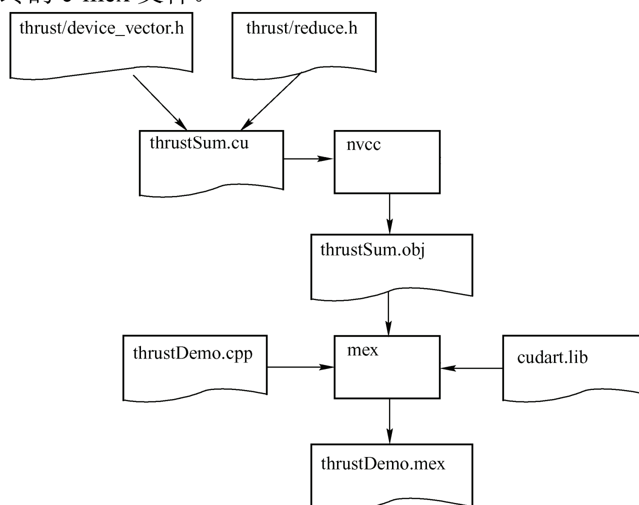


图 6.12　thrustDemo c-mex 创建过程

第 7 章　计算机图形学实例

7.1　本章学习目标

本章将介绍一种简单的计算机图形学算法：Marching Cubes。这种算法尽管简单，但却非常依赖数据集的数据密集运算。

本章中，你可以了解以下内容：

- Marching Cubes 算法介绍。
- 在 MATLAB 中实现此算法。
- 在 MATLAB 中提升速度。
- 利用 c-mex 和 CUDA 提升速度。

7.2　Marching Cubes 算法

Marching Cubes 是一种简单，但功能非常强大的计算机图形学算法，而且应用广泛，尤其是三维可视化中。该算法以三角形的形式，从给定的一组图中提取三维等值面，在这个等值面中每个点有相同的常数值。输入数据的来源众多，比如 CT 图和核磁共振（MRI）扫描图。输入数据通常源自一组二维图像，如图 7.1 所示。在体数据中，每个数据点称为一个体素（voxel）。图 7.2 中 8 个体素构成了一个立方体。

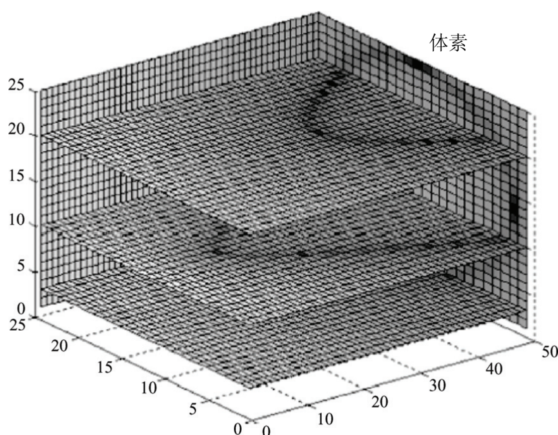


图 7.1　切片形式显示的采样体数据

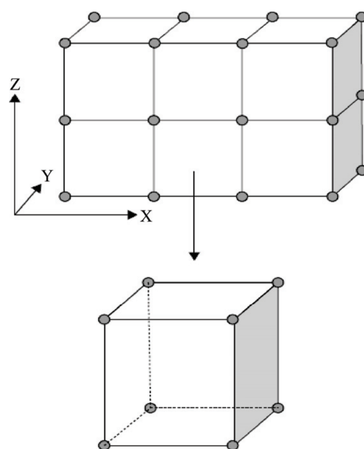


图 7.2　体数据、体素和立方体网格

如果有 $M\times N\times P$ 个体数据，则在整个数据集中会有$(M\text{-}1)\times(N\text{-}1)\times(P\text{-}1)$个立方体。每个立方体的顶点会分配一个体素值，每个顶点值可能高于或低于等值面的值。这个算法的基本思想就是找到等值面在立方体边界的交点，然后在这些交点中构造三角形。考虑图 7.2 中的立方体，并且标记每个顶点和棱。如图 7.3 中那样标记 8 个顶点和 12 条棱。因为立方体中每个顶点可以高于或低于等值面的值，所以共有 2^8=256 种可能的情况，由于有对称的情况，所以只有 15 种不同的情况，如图 7.4 所示。

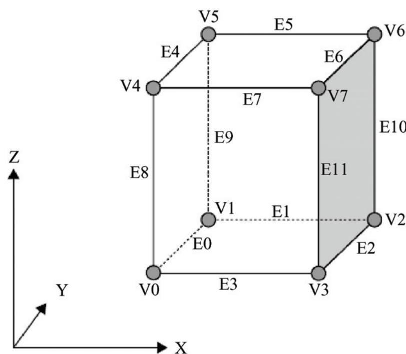


图 7.3　有顶点和棱坐标的立方体

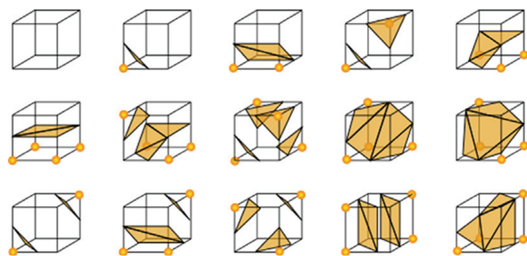


图 7.4　15 种不同的情况

为简单起见，假设顶点 V0 高于等值面值，其余点低于等值面值，所以 V0 是内部对象，如图 7.5 所示，这正是图 7.4 中的第二种情况。这种情况中，等值面穿过三条棱：E0、E8 和 E3，这三条棱的交点可线性插值。

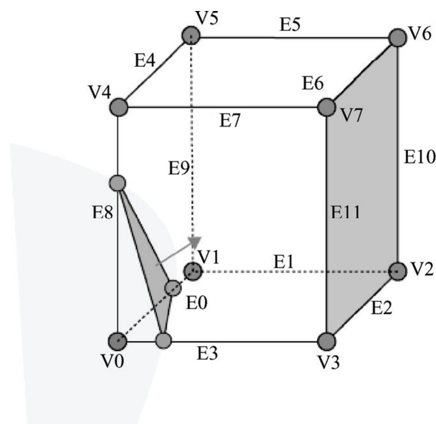


图 7.5　顶点 V0 大于等值面的值，生成一个带法向量的三角形

这个立方体的等值面类似于三角形。定义三角形的法向量，使其指向低于等值面的一侧。我们可以计算出所有其他的 255 种情况，并确定每种情况下的三角形。可以提前将这些值计算出来，然后用已经做好的查找表查找每一种情况。

这种算法提供了两种主要的表格：棱表格和三角形表格。首先，把 8 个顶点表示为 8 位二进制数，确定立方体数：

$$\text{第 } N \text{ 个比特 } = \begin{cases} 0 & \text{如果 Vi 处的值小于等值面的值} \\ 1 & \text{如果 Vi 处的值大于等于等值面的值} \end{cases}$$

本例中，立方体数是 1：

第 8 位	第 7 位	第 6 位	第 5 位	第 4 位	第 3 位	第 2 位	第 1 位
V7	V6	V5	V4	V3	V2	V1	V0
0	0	0	0	0	0	0	1

通过这个值，来看一下棱表格，从棱表格中可得到值为 265，如果把这个值转换成 12 位的二进制数，每位代表棱的值，如果等值面和棱相交，就设为 1。在本例中：

第12位	第11位	第10位	第9位	第8位	第7位	第6位	第5位	第4位	第3位	第2位	第1位
E11	E10	E9	E8	E7	E6	E5	E4	E3	E2	E1	E0
0	0	0	1	0	0	0	0	1	0	0	1

从二进制的值中，可以知道棱 E0、E3 和 E8 与等值面相交，在等值面和这些棱的交点处可以线性插值。比如在棱 E3 的交点可以估值为：

$$P3 = \frac{\text{isovalue} - I0}{I3 - I0} \times (V3 - V0) + V0$$

其中，I0 和 I3 分别是顶点 V0 和 V3 的强度值。

然后，第二个表格按顺序给出了一个三角形的索引，这能够确定法向量的方向（或者哪个面朝外）。对于本范例中的立方体，三角形表格给出：

0，8，3，-1，-1，-1，-1，-1，-1，-1，-1，-1，-1，-1，-1，-1

```
edgeTable = [
       0,  265,  515,  778, 1030, 1295, 1541, 1804, ...
    2060, 2309, 2575, 2822, 3082, 3331, 3593, 3840, ...
     400,  153,  915,  666, 1430, 1183, 1941, 1692, ...
    2460, 2197, 2975, 2710, 3482, 3219, 3993, 3728, ...
     560,  825,   51,  314, 1590, 1855, 1077, 1340, ...
…
…
    1804, 1541, 1295, 1030,  778,  515,  265,    0];


triTable = [
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     0,  8,  3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     0,  1,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     1,  8,  3,  9,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     1,  2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     0,  8,  3,  1,  2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     9,  2, 10,  0,  2,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     2,  8,  3,  2, 10,  8, 10,  9,  8, -1, -1, -1, -1, -1, -1, -1;
     3, 11,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;

…
…
     1, 10,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     1,  3,  8,  9,  1,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     0,  9,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     0,  3,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1];
```

图 7.6　棱和三角形查找表

按照表中的顺序，三角形是由点 E0、E8 和 E3 确定的。在表 7.6 中可以很容易查找出所有可能的三角形及对应的索引。

7.3 MATLAB 实现

本节介绍在 MATLAB 中怎样实现 Marching Cubes 算法。首先为了介绍这个算法，先在 MATLAB 中直接运行这个算法。在运行过程中，访问每个体素并确定所有的三角形。在 testSurfaceNoOpt.m 和 getSurfaceNoOpt.m，或在步骤的最后，有完整的执行过程。

7.3.1 步骤 1

首先，用 MATLAB 命令生成一个小的体数据样本 flow。在 MATLAB 命令窗口中，输入以下代码：

```
% generate sample volume data
[X, Y, Z, V] = flow;

% visualize volume data
figure
xmin = min(X(:));
ymin = min(Y(:));
zmin = min(Z(:));
xmax = max(X(:));
ymax = max(Y(:));
zmax = max(Z(:));
hslice = surf(linspace(xmin,xmax,100),  linspace(ymin,ymax,100),  zeros
(100));
rotate(hslice,[-1,0,0],-45)
xd = get(hslice,'XData');
yd = get(hslice,'YData');
zd = get(hslice,'ZData');
delete(hslice)
h = slice(X,Y,Z,V,xd,yd,zd);
set(h,'FaceColor','interp','EdgeColor','none','DiffuseStrength',0.8)
hold on
hx = slice(X,Y,Z,V,xmax,[],[]);
set(hx,'FaceColor', 'interp', 'EdgeColor', 'none')
hy = slice(X,Y,Z,V,[],ymax,[]);
set(hy,'FaceColor', 'interp', 'EdgeColor', 'none')
hz = slice(X,Y,Z,V,[],[],zmin);
set(hz,'FaceColor', 'interp', 'EdgeColor', 'none')
```

默认情况下，会生成四个变量：X、Y、Z 和 V，它们是包含体素位置和强度

的 25×25×50 的矩阵。其余代码会进行数据的可视化。如果现在执行代码，会生成如图 7.7 所示的体数据。

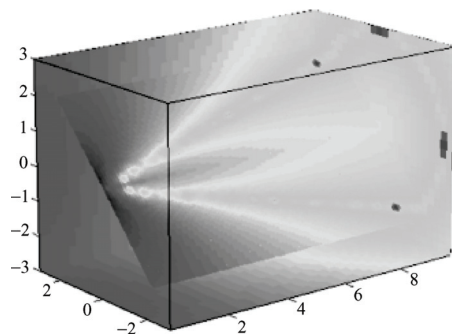


图 7.7 体数据样本

如果想进一步了解可视化操作，可见 MATLAB > User's Guide > 3-D Visualization > Volume Visualization Techniques > Exploring Volumes with Slice Planes 中的 MATLAB 帮助，就可以知道沿着 X、Y、Z 方向的体数据大小，并定义等值面值为-3。

7.3.2 步骤 2

定义函数来执行 Marching Cubes 算法，并保存为 getSurfaceNoOpt.m。这个函数有五个输入和两个输出返回值，包含体数据中三角形的所有顶点和索引。这里只显示部分表值。全部表值可以在给出的 getSurfaceNoOpt.m 文件中找到。

```
function [Vertices, Indices] = getSurfaceNoOpt(X, Y, Z, V, isovalue)

edgeTable = uint16([
       0,  265,  515,  778,  1030,  1295,  1541,  1804, ...
    2060,  2309,  2575,  2822,  3082,  3331,  3593,  3840, ...
    ...
    ...
    1804, 1541, 1295, 1030, 778, 515, 265,  0]);

triTable = [
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
    ...
    ...
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1];

sizeX = size(V, 1);
sizeY = size(V, 2);
sizeZ = size(V, 3);
```

```
MAX_VERTICES = 15;

% vertices and indices
VertexBin = single(zeros(sizeX, sizeY, sizeZ, MAX_VERTICES, 3));
TriCounter = zeros(sizeX, sizeY, sizeZ);
totalTriangles = 0;
```

首先为 Marching Cubes 算法创建两个预先计算的查找表。接下来，确定体数据沿 *X*、*Y* 和 *Z* 方向的大小。然后为所有顶点和索引存储预分配临时变量。

7.3.3　步骤 3

在这一步中，利用循环嵌套的方式访问体数据中的所有立方体。如果体数据大小为 $M \times N \times P$，则共有 $(M-1) \times (N-1) \times (P-1)$ 个立方体。

```
for k = 1:sizeZ - 1
    for j = 1:sizeY - 1
        for i = 1:sizeX - 1
        ...

         % algorithm goes in here
         ...
         end
     end
 end
```

7.3.4　步骤 4

在最内层的循环中，读取正在处理的立方体的 8 个顶点的位置和强度值。如图 7.3 所示，根据顶点标号进行分配。

```
% temp. vars for voxels and isopoints.
voxels = single(zeros(8, 4));  %[v, x, y, z]
isoPos = single(zeros(12, 3));  %[x, y, z]

%cube
voxels(1,:) = [V(i,     j,     k),     X(i,     j,     k),     Y(i,     j,     k),
               Z(i,     j,     k)];
voxels(2,:) = [V(i,     j + 1, k),     X(i,     j + 1, k),     Y(i,     j + 1, k),
               Z(i,     j + 1, k)];
voxels(3,:) = [V(i + 1, j + 1, k),     X(i + 1, j + 1, k),     Y(i + 1, j + 1, k),
               Z(i + 1, j + 1, k)];
voxels(4,:) = [V(i + 1, j,     k),     X(i + 1, j,     k),     Y(i + 1, j,     k),
               Z(i + 1, j,     k)];
voxels(5,:) = [V(i,     j,     k + 1), X(i,     j,     k + 1), Y(i,     j,     k + 1),
               Z(i,     j,     k + 1)];
voxels(6,:) = [V(i,     j + 1, k + 1), X(i,     j + 1, k + 1), Y(i,     j + 1, k + 1),
               Z(i,     j + 1, k + 1)];
```

```
voxels(7,:)=[V(i+1, j+1, k+1), X(i+1, j+1, k+1), Y(i+1, j+1, k+1),
            Z(i+1, j+1, k+1)];
voxels(8,:)=[V(i+1, j,   k+1), X(i+1, j,   k+1), Y(i+1, j,   k+1),
            Z(i+1, j,   k+1)];
```

7.3.5 步骤 5

检查每个顶点的强度值是否大于或者小于等值面值。如果这个值大于等值面值，该顶点的 1 比特位置值将设定为 1。因为有 8 个顶点，所以总共有 256 种情况。检测 8 个顶点的所有值后，确定立方体索引。利用这个索引，从棱表格中获取棱数值。

```
% find the cube index
cubeIndex = uint16(0);
for n = 1:8
    if voxels(n, 1) >= isovalue
        cubeIndex = bitset(cubeIndex, n);
    end
end
cubeIndex = cubeIndex + 1;

% get edges from edgeTable
edges = edgeTable(cubeIndex);
if edges == 0
    continue;
end
```

7.3.6 步骤 6

从棱表格中获取棱值之后，可以确定等值面与棱相交的位置，并查找这些棱的交点。首先，创建一个新的内插函数。这个函数以体素值和等值面值作为输入，以内插位置为输出。

```
function [varargout] = interpolatePos(isovalue, voxel1, voxel2)
    scale = (isovalue - voxel1(:,1)) ./ (voxel2(:,1) - voxel1(:,1));
    interpolatedPos = voxel1(:,2:4) + [scale .* (voxel2(:,2) - voxel1(:,2)), ...
                                       scale .* (voxel2(:,3) - voxel1(:,3)), ...
                                       scale .* (voxel2(:,4) - voxel1(:,4))];

    if nargout == 1 || nargout == 0
        varargout{1} = interpolatedPos;
    elseif nargout == 3
        varargout{1} = interpolatedPos(:,1);
        varargout{2} = interpolatedPos(:,2);
        varargout{3} = interpolatedPos(:,3);
    end
```

回到主函数，继续检测棱值，如果可行，在每个棱调用内插函数：

```
% check 12 edges
if bitand(edges, 1)
    isoPos(1,:) = interpolatePos(isovalue, voxels(1,:), voxels(2,:));
end
if bitand(edges, 2)
    isoPos(2,:) = interpolatePos(isovalue, voxels(2,:), voxels(3,:));
end
if bitand(edges, 4)
    isoPos(3,:) = interpolatePos(isovalue, voxels(3,:), voxels(4,:));
end
if bitand(edges, 8)
    isoPos(4,:) = interpolatePos(isovalue, voxels(4,:), voxels(1,:));
end
if bitand(edges, 16)
    isoPos(5,:) = interpolatePos(isovalue, voxels(5,:), voxels(6,:));
end
if bitand(edges, 32)
    isoPos(6,:) = interpolatePos(isovalue, voxels(6,:), voxels(7,:));
end
if bitand(edges, 64)
    isoPos(7,:) = interpolatePos(isovalue, voxels(7,:), voxels(8,:));
end
if bitand(edges, 128)
    isoPos(8,:) = interpolatePos(isovalue, voxels(8,:), voxels(5,:));
end
if bitand(edges, 256)
    isoPos(9,:) = interpolatePos(isovalue, voxels(1,:), voxels(5,:));
end
if bitand(edges, 512)
    isoPos(10,:) = interpolatePos(isovalue, voxels(2,:), voxels(6,:));
end
if bitand(edges, 1024)
    isoPos(11,:) = interpolatePos(isovalue, voxels(3,:), voxels(7,:));
end
if bitand(edges, 2048)
    isoPos(12,:) = interpolatePos(isovalue, voxels(4,:), voxels(8,:));
end
```

7.3.7　步骤 7

从三角形表格中，可以知道有多少个三角形及它们的索引。仔细检查三角形表格中预先设定的顶点顺序，并记录所有确定的三角形，最多有 5 个三角形。

```
%walk through the triTable and get the triangle(s) vertices
numTriangles = 0;
numVertices = 0;
for n = 1:3:16
```

```
    if triTable(cubeIndex, n) < 0
        break;
    end

    % first vertex
    numVertices = numVertices + 1;
    edgeIndex = triTable(cubeIndex, n) + 1;
    vertexBin(i, j, k, numVertices, :) = isoPos(edgeIndex, :);
    % second vertex
    numVertices = numVertices + 1;
    edgeIndex = triTable(cubeIndex, n + 1) + 1;
    vertexBin(i, j, k, numVertices, :) = isoPos(edgeIndex, :);
    % third vertex
    numVertices = numVertices + 1;
    edgeIndex = triTable(cubeIndex, n + 2) + 1;
    vertexBin(i, j, k, numVertices, :) = isoPos(edgeIndex, :);

    numTriangles = numTriangles + 1;
end
TriCounter(i, j, k) = numTriangles;
totalTriangels = totalTriangles + numTriangles;
```

7.3.8 步骤 8

在立方体中找到所有三角形后，把它们添加到顶点和索引的输出存储中：

```
Vertices = single(zeros(totalTriangles * 3, 3));
Indices = uint32(zeros(totalTriangels, 3));
vIdx = 0;
tIdx = 0;
for k = 1:sizeZ - 1
    for j = 1:sizeY - 1
        for i = 1:sizeX - 1

            count = TriCounter(i, j, k);
            if count < 1
               continue;
            end

            for t = 0:count - 1
                vIdx = vIdx + 1;
                Vertices(vIdx, :) = vertexBin(i, j, k, 3*t + 1, :);
                vIdx = vIdx + 1;
                Vertices(vIdx, :) = vertexBin(i, j, k, 3*t + 2, :);
                vIdx = vIdx + 1;
                Vertices(vIdx, :) = vertexBin(i, j, k, 3*t + 3, :);

                tIdx = tIdx + 1;
                Indices(tIdx, :) = uint32([3 * tIdx - 2, 3 * tIdx - 1, 3 * tIdx]);
            end
```

```
        end
    end
end
```

7.3.9　步骤 9

计算所有的顶点和三角形索引后，可以如下方式进行三角形可视化。回到 testSurfaceNoOpt.m，添加以下几行实现三角形可视化：

```
% data type to single
X = single(X);
Y = single(Y);
Z = single(Z);
V = single(V);
isovalue = single(-3);

% Marching cubes
[Vertices1, Indices1] = getSurfaceNoOpt(X, Y, Z, V, isovalue);

% visualize triangles
figure
p = patch('Faces', Indices1, 'Vertices', Vertices1);
set(p, 'FaceColor', 'none', 'EdgeColor', 'red');
daspect([1,1,1])
view(3);
camlight
lighting gouraud
grid on
```

利用 Marching Cubes 找到的三角形如图 7.8 所示。

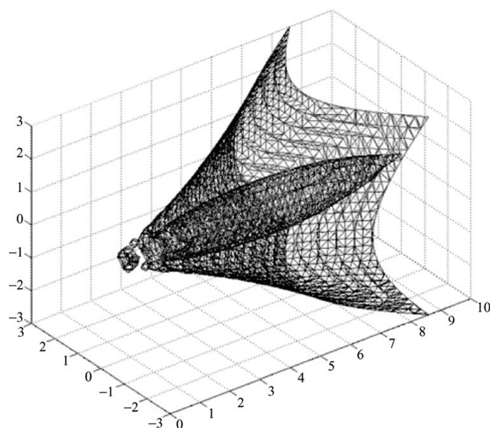


图 7.8　未优化且等值面值=−3 时的等值面

下面是在 MATLAB 中完整的实现过程，包括两个查找表。注意，在最后的结果中有重复的顶点。可以继续删除这些重复顶点，减小最终存储的大小。然而，出于展示算法的目的，我们可以到这结束，不进一步减少重复点。

这是三个文件完整的代码，所示表格值是截短的。全部表格值可以在提供的 getSurfaceNoOpt.m 文件中找到。

```
getSurfaceNoOpt.m
function [Vertices, Indices] = getSurfaceNoOpt(X, Y, Z, V, isovalue)

edgeTable = uint16([
       0,  265,  515,  778, 1030, 1295, 1541, 1804, ...
    2060, 2309, 2575, 2822, 3082, 3331, 3593, 3840, ...
     400,  153,  915,  666, 1430, 1183, 1941, 1692, ...
    2460, 2197, 2975, 2710, 3482, 3219, 3993, 3728, ...
     560,  825,   51,  314, 1590, 1855, 1077, 1340, ...
...
...
...
    3728, 3993, 3219, 3482, 2710, 2975, 2197, 2460, ...
    1692, 1941, 1183, 1430,  666,  915,  153,  400, ...
    3840, 3593, 3331, 3082, 2822, 2575, 2309, 2060, ...
    1804, 1541, 1295, 1030,  778,  515,  265,    0]);

triTable = [
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     0,  8,  3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     0,  1,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     1,  8,  3,  9,  8,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
...
...
...
     1,  3,  8,  9,  1,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     0,  9,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     0,  3,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1];

sizeX = size(V, 1);
sizeY = size(V, 2);
sizeZ = size(V, 3);

MAX_VERTICES = 15;

% vertices and indices
VertexBin = single(zeros(sizeX, sizeY, sizeZ, MAX_VERTICES, 3));
TriCounter = zeros(sizeX, sizeY, sizeZ);
totalTriangles = 0;
```

```
% marching through the cubes
for k = 1:sizeZ - 1
    for j = 1:sizeY - 1
        for i = 1:sizeX - 1

            % temp. vars for voxels and isopoints.
            voxels = single(zeros(8, 4)); %[v, x, y, z]
            isoPos = single(zeros(12, 3)); %[x, y, z]

            %cube
            voxels(1,:) = [V(i, j, k),  X(i, j, k), Y(i, j, k), Z(i, j, k)];
voxels(2,:) = [V(i, j+1,k),  X(i, j+1,k), Y(i, j+1,k),
              Z(i, j+1,k)];
voxels(3,:) = [V(i+1,j+1,k), X(i+1,j+1,k), Y(i+1,j+1,k),
             Z(i+1,j+1,k)];
voxels(4,:) = [V(i+1,j, k), X(i+1,j, k), Y(i+1,j, k),
              Z(i+1,j, k)];
voxels(5,:) = [V(i, j, k+1),X(i, j, k+1),Y(i, j, k+1),
              Z(i, j, k+1)];
voxels(6,:) = [V(i, j+1,k+1),X(i, j+1,k+1),Y(i, j+1,
              k+1),Z(i, j+1,k+1)];
voxels(7,:) = [V(i+1,j+1,k+1),X(i+1,j+1,k+1),Y(i+1,
              j+1,k+1),Z(i+1,j+1,k+1)];
voxels(8,:) = [V(i+1,j, k+1),X(i+1,j, k+1),Y(i+1,j,
              k+1),Z(i+1,j, k+1)];

% find the cube index
cubeIndex = uint16(0);
for n = 1:8
    if voxels(n, 1) >= isovalue
        cubeIndex = bitset(cubeIndex, n);
    end
end
cubeIndex = cubeIndex + 1;

% get edges from edgeTable
edges = edgeTable(cubeIndex);
if edges == 0
    continue;
end

% check 12 edges
if bitand(edges, 1)
    isoPos(1,:) = interpolatePos(isovalue, voxels(1,:),
                  voxels(2,:));
end
if bitand(edges, 2)
    isoPos(2,:) = interpolatePos(isovalue, voxels(2,:),
                  voxels(3,:));
```

```
end
if bitand(edges, 4)
    isoPos(3,:) = interpolatePos(isovalue, voxels(3,:),
                  voxels(4,:));
end
if bitand(edges, 8)
    isoPos(4,:) = interpolatePos(isovalue, voxels(4,:),
                  voxels(1,:));
end
if bitand(edges, 16)
    isoPos(5,:) = interpolatePos(isovalue, voxels(5,:),
                  voxels(6,:));
end
if bitand(edges, 32)
    isoPos(6,:) = interpolatePos(isovalue, voxels(6,:),
                  voxels(7,:));
end
if bitand(edges, 64)
    isoPos(7,:) = interpolatePos(isovalue, voxels(7,:),
                  voxels(8,:));
end
if bitand(edges, 128)
    isoPos(8,:) = interpolatePos(isovalue, voxels(8,:),
                  voxels(5,:));
end
if bitand(edges, 256)
    isoPos(9,:) = interpolatePos(isovalue, voxels(1,:),
                  voxels(5,:));
end
if bitand(edges, 512)
    isoPos(10,:) = interpolatePos(isovalue, voxels(2,:),
                   voxels(6,:));
end
if bitand(edges, 1024)
    isoPos(11,:) = interpolatePos(isovalue, voxels(3,:),
                   voxels(7,:));
end
if bitand(edges, 2048)
    isoPos(12,:) = interpolatePos(isovalue, voxels(4,:),
                   voxels(8,:));
end

%walk through the triTable and get the triangle(s) vertices
numTriangles = 0;
numVertices = 0;
for n = 1:3:16
```

```
    if triTable(cubeIndex, n) < 0
        break;
    end

    % first vertex
    numVertices = numVertices + 1;
    edgeIndex = triTable(cubeIndex, n) + 1;
    vertexBin(i, j, k, numVertices, :) = isoPos(edgeIndex, :);
    % second vertex
    numVertices = numVertices + 1;
    edgeIndex = triTable(cubeIndex, n + 1) + 1;
    vertexBin(i, j, k, numVertices, :) = isoPos(edgeIndex, :);
    % third vertex
    numVertices = numVertices + 1;
    edgeIndex = triTable(cubeIndex, n + 2) + 1;
    vertexBin(i, j, k, numVertices, :) = isoPos(edgeIndex, :);

                numTriangles = numTriangles + 1;
            end
            TriCounter(i, j, k) = numTriangles;
            totalTriangels = totalTriangles + numTriangles;
        end
    end
end

Vertices = single(zeros(totalTriangles * 3, 3));
Indices = uint32(zeros(totalTriangels, 3));
vIdx = 0;
tIdx = 0;
for k = 1:sizeZ - 1
    for j = 1:sizeY - 1
        for i = 1:sizeX - 1

            count = TriCounter(i, j, k);
            if count < 1
                continue;
            end

            for t = 0:count - 1
                vIdx = vIdx + 1;
                Vertices(vIdx, :) = vertexBin(i, j, k, 3*t + 1, :);
                vIdx = vIdx + 1;
                Vertices(vIdx, :) = vertexBin(i, j, k, 3*t + 2, :);
                vIdx = vIdx + 1;
                Vertices(vIdx, :) = vertexBin(i, j, k, 3*t + 3, :);
                tIdx = tIdx + 1;
                Indices(tIdx, :) = uint32([3 * tIdx - 2, 3 * tIdx - 1, 3 * tIdx]);
            end
        end
```

```
    end
end
```

InterpolatePos.m

```
function [varargout] = interpolatePos(isovalue, voxel1, voxel2)
    scale = (isovalue - voxel1(:,1)) ./ (voxel2(:,1) - voxel1(:,1));
    interpolatedPos = voxel1(:,2:4) + [scale .* (voxel2(:,2) - voxel1(:,2)), ...
                                       scale .* (voxel2(:,3) - voxel1(:,3)), ...
                                       scale .* (voxel2(:,4) - voxel1(:,4))];

if nargout == 1 || nargout == 0
    varargout{1} = interpolatedPos;
elseif nargout == 3
    varargout{1} = interpolatedPos(:,1);
    varargout{2} = interpolatedPos(:,2);
    varargout{3} = interpolatedPos(:,3);
end
```

testSurfaceNoOpt.m

```
% generate sample volume data
[X, Y, Z, V] = flow;

% visualize volume data
figure
xmin = min(X(:));
ymin = min(Y(:));
zmin = min(Z(:));
xmax = max(X(:));
ymax = max(Y(:));
zmax = max(Z(:));
hslice = surf(linspace(xmin,xmax,100), linspace(ymin,ymax,100), zeros(100));
rotate(hslice,[-1,0,0],-45)
xd = get(hslice,'XData');
yd = get(hslice,'YData');
zd = get(hslice,'ZData');
delete(hslice)
h = slice(X,Y,Z,V,xd,yd,zd);
set(h,'FaceColor', 'interp', 'EdgeColor', 'none', 'DiffuseStrength', 0.8)
hold on
hx = slice(X,Y,Z,V,xmax,[],[]);
set(hx,'FaceColor', 'interp', 'EdgeColor', 'none')
hy = slice(X,Y,Z,V,[],ymax,[]);
set(hy,'FaceColor', 'interp', 'EdgeColor', 'none')
hz = slice(X,Y,Z,V,[],[],zmin);
set(hz,'FaceColor', 'interp', 'EdgeColor', 'none')

% data type to single
X = single(X);
Y = single(Y);
```

```
Z = single(Z);
V = single(V);
isovalue = single(-3);

% Marching cubes
[Vertices1, Indices1] = getSurfaceNoOpt(X, Y, Z, V, isovalue);

% visualize triangles
figure
p = patch('Faces', Indices1, 'Vertices', Vertices1);
set(p, 'FaceColor', 'none', 'EdgeColor', 'red');
daspect([1,1,1])
view(3);
camlight
lighting gouraud
grid on
```

7.3.10　时间分析

现在，可以利用 MATLAB 的 Profiler 分析执行情况。按〈Ctrl+5〉键或者从 MATLAB 主菜单中选择 Windows > Profiler 打开 Profiler 窗口。然后，输入 MATLAB 代码文件名并运行它，得到如图 7.9 所示的分析总结。

Profiler

File Edit Debug Desktop Window Help

Start Profiling　Run this code: testSurfaceNoOpt　Profile time: 2 sec

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
axescheck	7	0.019 s	0.004 s	
camlight	1	0.007 s	0.002 s	
camlight>righthanded	1	0 s	0.000 s	
camrotate	1	0.005 s	0.005 s	
camrotate>crossSimple	2	0 s	0.000 s	
cart2sph	1	0.007 s	0.007 s	
caxis	4	0.003 s	0.003 s	
cla	3	0.041 s	0.000 s	
daspect	1	0.001 s	0.001 s	
deal	8	0.001 s	0.001 s	
double.superiorfloat	2	0 s	0.000 s	
findall	3	0.002 s	0.000 s	
...set(rootobj,'ShowHiddenHandles',Temp)	3	0 s	0.000 s	
...set(rootobj,'ShowHiddenHandles',Temp)	3	0 s	0.000 s	
...all>showHiddenHandlesToFindAllHandles	3	0 s	0.000 s	
findobj	5	0.002 s	0.000 s	
flow	1	0.020 s	0.007 s	
getSurfaceNoOpt	1	1.946 s	1.640 s	
graph2dhelper	6	0 s	0.000 s	
graph3d.surfaceplot.surfaceplot	2	0.013 s	0.001 s	
...surfaceplot.surfaceplot>LGetListeners	3	0 s	0.000 s	
...surfaceplot.surfaceplot>LSetListeners	2	0 s	0.000 s	
...surfaceplot.surfaceplot>LaddListeners	2	0 s	0.000 s	

图 7.9　未优化的 MATLAB 代码时间分析

总的来说，共需要大约 1.9 s。点击 Profiler 窗口中的 getSurfaceNoOpt 链接，可以看到更多细节结果，如图 7.10 所示。

根据分析结果，代码在分配空体素和查找立方体索引中耗费了大多数时间。

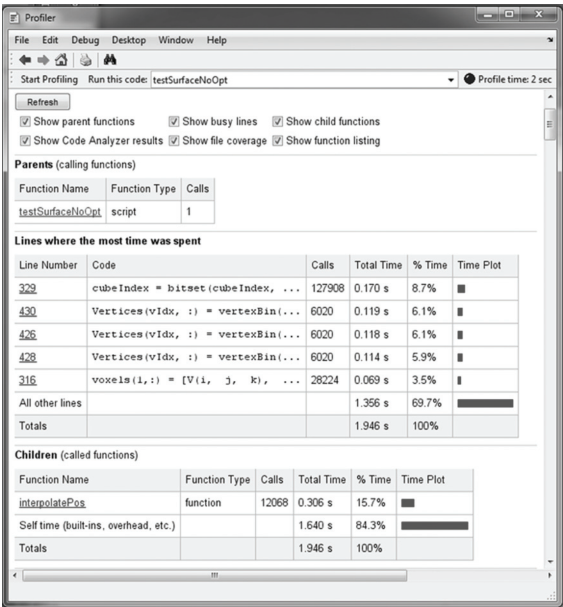


图 7.10　时间分析细节

7.4　采用 CUDA 和 c-mex 实现算法

直接执行 Marching Cubes 算法，大约需要 1.9 s 的时间。本节将展示如何只利用 MATLAB 代码实现性能改善。采用第 1 章介绍的向量化和预分配的概念。我们的策略不是访问每个体素，而是预先分配临时内存来存储中间结果，并在不引入三重嵌套循环的情况下计算三角形。

7.4.1　步骤 1

创建一个新的函数 getSurfaceWithOpt，输入如下代码，并将其保存为 getSurfaceWithOpt.m：

```
getSurfaceWithOpt.m.
getSurfaceWithOpt.m
function [Vertices, Indices] = getSurfaceWithOpt(X, Y, Z, V, isovalue)

edgeTable = uint16([
        0,  265,  515,  778, 1030, 1295, 1541, 1804, ...
    2060, 2309, 2575, 2822, 3082, 3331, 3593, 3840, ...
    ...
    ...
    1804, 1541, 1295, 1030,778,515,265,0]);
```

```
triTable = [
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
     0,8,3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1;
    ...
    ...
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1];

sx = size(V, 1);
sy = size(V, 2);
sz = size(V, 3);

% Cube Index
CV = uint16(V > isovalue);
Cubes = zeros(sx-1, sy-1, sz-1, 'uint16');
Cubes = Cubes + CV(1:sx-1, 1:sy-1, 1:sz-1);
Cubes = Cubes + CV(1:sx-1, 2:sy, 1:sz-1) * 2;
Cubes = Cubes + CV(2:sx, 2:sy, 1:sz-1) * 4;
Cubes = Cubes + CV(2:sx, 1:sy-1, 1:sz-1) * 8;
Cubes = Cubes + CV(1:sx-1, 1:sy-1, 2:sz) * 16;
Cubes = Cubes + CV(1:sx-1, 2:sy, 2:sz) * 32;
Cubes = Cubes + CV(2:sx, 2:sy, 2:sz) * 64;
Cubes = Cubes + CV(2:sx, 1:sy-1, 2:sz) * 128;
Cubes = Cubes + 1;

% Edges
Edges = edgeTable(Cubes);
EdgeIdx = find(Edges > 0);
EdgeVal = Edges(EdgeIdx);

% Vertices with edges
[vdx, vdy, vdz] = ind2sub(size(Edges), EdgeIdx);
idx = sub2ind(size(X), vdx, vdy, vdz);
Vt1 = [V(idx), X(idx), Y(idx), Z(idx)];
idx = sub2ind(size(X), vdx, vdy + 1, vdz);
Vt2 = [V(idx), X(idx), Y(idx), Z(idx)];
idx = sub2ind(size(X), vdx + 1, vdy + 1, vdz);
Vt3 = [V(idx), X(idx), Y(idx), Z(idx)];
idx = sub2ind(size(X), vdx + 1, vdy, vdz);
Vt4 = [V(idx), X(idx), Y(idx), Z(idx)];
idx = sub2ind(size(X), vdx, vdy, vdz + 1);
Vt5 = [V(idx), X(idx), Y(idx), Z(idx)];

idx = sub2ind(size(X), vdx, vdy + 1, vdz + 1);
Vt6 = [V(idx), X(idx), Y(idx), Z(idx)];
idx = sub2ind(size(X), vdx + 1, vdy + 1, vdz + 1);
Vt7 = [V(idx), X(idx), Y(idx), Z(idx)];
idx = sub2ind(size(X), vdx + 1, vdy, vdz + 1);
Vt8 = [V(idx), X(idx), Y(idx), Z(idx)];
```

```
% EdgeNumber
PosX = zeros(size(EdgeVal,1), 12, 'single');
PosY = zeros(size(EdgeVal,1), 12, 'single');
PosZ = zeros(size(EdgeVal,1), 12, 'single');
idx = find(uint16(bitand(EdgeVal, 1)));
[PosX(idx,1), PosY(idx,1), PosZ(idx,1)] = interpolatePos(isovalue, Vt1
(idx,:), Vt2(idx,:));
idx = find(uint16(bitand(EdgeVal, 2)));
[PosX(idx,2), PosY(idx,2), PosZ(idx,2)] = interpolatePos(isovalue, Vt2
(idx,:), Vt3(idx,:));
idx = find(uint16(bitand(EdgeVal, 4)));
[PosX(idx,3), PosY(idx,3), PosZ(idx,3)] = interpolatePos(isovalue, Vt3
(idx,:), Vt4(idx,:));
idx = find(uint16(bitand(EdgeVal, 8)));
[PosX(idx,4), PosY(idx,4), PosZ(idx,4)] = interpolatePos(isovalue, Vt4
(idx,:), Vt1(idx,:));
idx = find(uint16(bitand(EdgeVal, 16)));
[PosX(idx,5), PosY(idx,5), PosZ(idx,5)] = interpolatePos(isovalue, Vt5
(idx,:), Vt6(idx,:));
idx = find(uint16(bitand(EdgeVal, 32)));
[PosX(idx,6), PosY(idx,6), PosZ(idx,6)] = interpolatePos(isovalue, Vt6
(idx,:), Vt7(idx,:));
idx = find(uint16(bitand(EdgeVal, 64)));
[PosX(idx,7), PosY(idx,7), PosZ(idx,7)] = interpolatePos(isovalue, Vt7
(idx,:), Vt8(idx,:));
idx = find(uint16(bitand(EdgeVal, 128)));
[PosX(idx,8), PosY(idx,8), PosZ(idx,8)] = interpolatePos(isovalue, Vt8
(idx,:), Vt5(idx,:));
idx = find(uint16(bitand(EdgeVal, 256)));
[PosX(idx,9), PosY(idx,9), PosZ(idx,9)] = interpolatePos(isovalue, Vt1
(idx,:), Vt5(idx,:));
idx = find(uint16(bitand(EdgeVal, 512)));
[PosX(idx,10), PosY(idx,10), PosZ(idx,10)] = interpolatePos(isovalue,
Vt2(idx,:), Vt6(idx,:));
idx = find(uint16(bitand(EdgeVal, 1024)));
[PosX(idx,11), PosY(idx,11), PosZ(idx,11)] = interpolatePos(isovalue,
Vt3(idx,:), Vt7(idx,:));
idx = find(uint16(bitand(EdgeVal, 2048)));
[PosX(idx,12), PosY(idx,12), PosZ(idx,12)] = interpolatePos(isovalue,
Vt4(idx,:), Vt8(idx,:));

% Triangles
Vertices = zeros(0, 3, 'single');
TriVal = triTable(Cubes(EdgeIdx),:) + 1;

for i = 1:3:15
    idx = find(TriVal(:,i) > 0);
    if isempty(idx)
        continue;
```

```
        end
        TriVtx = TriVal(idx, i:i + 2);
        vx1 = PosX(sub2ind(size(PosX), idx, TriVtx(:,1)));
        vy1 = PosY(sub2ind(size(PosY), idx, TriVtx(:,1)));
        vz1 = PosZ(sub2ind(size(PosZ), idx, TriVtx(:,1)));
        vx2 = PosX(sub2ind(size(PosX), idx, TriVtx(:,2)));
        vy2 = PosY(sub2ind(size(PosY), idx, TriVtx(:,2)));
        vz2 = PosZ(sub2ind(size(PosZ), idx, TriVtx(:,2)));
        vx3 = PosX(sub2ind(size(PosX), idx, TriVtx(:,3)));
        vy3 = PosY(sub2ind(size(PosY), idx, TriVtx(:,3)));
        vz3 = PosZ(sub2ind(size(PosZ), idx, TriVtx(:,3)));
        vsz = 3 * size(vx1, 1);
        t1 = zeros(vsz, 3, 'single');
        t1(1:3:vsz,:) = [vx1 vy1 vz1];
        t1(2:3:vsz,:) = [vx2 vy2 vz2];
        t1(3:3:vsz,:) = [vx3 vy3 vz3];
        Vertices = [Vertices; t1];
    end

    Indices = reshape(1:size(Vertices,1), 3, size(Vertices,1)/3)';
```

在这个代码中，没有循环嵌套，而是将顶点和索引作为一个整体加以计算，这是以存放更多临时变量为代价的。随着体数据的增加，这个方法可能变得不可行。无论如何，如果执行这段代码，我们会惊讶于它所能节省的时间。

7.4.2　步骤 2

修改 testSurfaceNoOpt.m 文件如下，并保存为 testSurfaceWithOpt.m：

```
testSurfaceWithOpt.m
% generate sample volume data
[X, Y, Z, V] = flow;
X = single(X);
Y = single(Y);
Z = single(Z);
V = single(V);
isovalue = single(-3);

[Vertices2, Indices2] = getSurfaceWithOpt(X, Y, Z, V, isovalue);

% visualize triangles
figure
p = patch('Faces', Indices2, 'Vertices', Vertices2);
set(p, 'FaceColor', 'none', 'EdgeColor', 'green');
daspect([1,1,1])
view(3);
camlight
lighting gouraud
grid on
```

运行这个脚本后，会得到相同的输出数据，仅三角形以不同的顺序存储。但是，基本上会得到和之前一样的三角形，如图 7.11 所示。

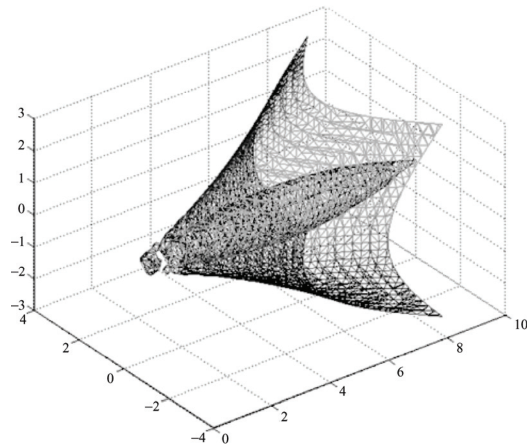
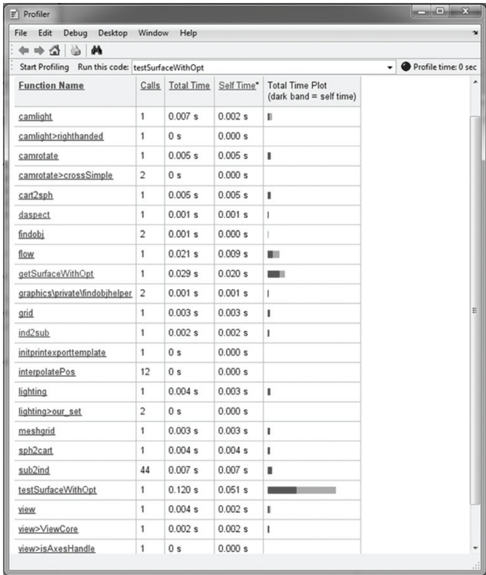


图 7.11　优化后等值面值为−3 时的等值面

7.4.3　时间分析

现在，进行新的优化后代码的时间分析。回到 Profiler 并运行 testSurfaceWithOpt.m，如图 7.12 所示。



Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
camlight	1	0.007 s	0.002 s	
camlight>righthanded	1	0 s	0.000 s	
camrotate	1	0.005 s	0.005 s	
camrotate>crossSimple	2	0 s	0.000 s	
cart2sph	1	0.005 s	0.005 s	
daspect	1	0.001 s	0.001 s	
findobj	2	0.001 s	0.000 s	
flow	1	0.021 s	0.009 s	
getSurfaceWithOpt	1	0.029 s	0.020 s	
graphics\private\findobjhelper	2	0.001 s	0.001 s	
grid	1	0.003 s	0.003 s	
ind2sub	1	0.002 s	0.002 s	
initprintexporttemplate	1	0 s	0.000 s	
interpolatePos	12	0 s	0.000 s	
lighting	1	0.004 s	0.003 s	
lighting>our_set	2	0 s	0.000 s	
meshgrid	1	0.003 s	0.003 s	
sph2cart	1	0.004 s	0.004 s	
sub2ind	44	0.007 s	0.007 s	
testSurfaceWithOpt	1	0.120 s	0.051 s	
view	1	0.004 s	0.002 s	
view>ViewCore	1	0.002 s	0.002 s	
view>isAxesHandle	1	0 s	0.000 s	

图 7.12　优化后的 MATLAB 代码时间分析

此时，与没有优化时的 1.9 s 相比，新函数 getSurfaceWithOpt 只用了 0.029 s。与直接实现相比，有了很大的改善。通过去除三重嵌套循环，可以节省大部分时间。

7.5　用 c-mex 函数和 GPU 实现

即使在没有引入 GPU 和 c-mex 函数的情况下，如 7.4 节那样，利用向量化和预分配技术精心设计代码，也可以在速度方面得到巨大的改善。本节进一步探索利用 c-mex 函数和 GPU 能获得多大的速度提升。总体来说，遵从第一种没有优化的代码，用 CUDA 函数代替所有内层循环操作。

7.5.1　步骤 1

创建 c-mex 函数的子例行程序，并以如下代码填充主函数主体部分。保存为 getSurfaceCuda.cpp：

```
getSurfaceCuda.cpp
#include "mex.h"
#include <vector>
#include <cuda_runtime.h>
#include "MarchingCubes.h"

// [Vertices, Indices] = getSurface(X, Y, Z, V, isovalue)
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    if (nrhs != 5)
        mexErrMsgTxt("Invaid number of input arguments");

    if (nlhs != 2)
        mexErrMsgTxt("Invalid number of outputs");
    if (!mxIsSingle(prhs[0]) && !mxIsSingle(prhs[1]) &&
        !mxIsSingle(prhs[2]) && !mxIsSingle(prhs[3]) &&
        !mxIsSingle(prhs[4]))
        mexErrMsgTxt("input vector data type must be single");

    const mwSize* size = mxGetDimensions(prhs[3]);
    int sizeX = size[0];
    int sizeY = size[1];
    int sizeZ = size[2];

    float* X = (float*mxGetData(prhs[0]);
    float*Y = (float*mxGetData(prhs[1]);
    float*Z = (float*mxGetData(prhs[2]);
    float*V = (float*mxGetData(prhs[3]);
    float isovalue = *float*mxGetData(prhs[4]));

    float*vertices[3];
    unsigned int*indices[3];
    int numVertices = 0;
    int numTriangles = 0;
```

```
    marchingCubes(isovalue,
                  X, Y, Z, V,
                  sizeX, sizeY, sizeZ,
                  vertices, indices,
                  numVertices, numTriangles);
    cudaError_t error = cudaGetLastError();
    if (error != cudaSuccess)
    {
       mexPrintf("%s\n", cudaGetErrorString(error));
       mexErrMsgTxt("CUDA failed\n");
    }

    mexPrintf("numVertices = %d\n", numVertices);
    mexPrintf("numTriangles = %d\n", numTriangles);

    plhs[0] = mxCreateNumericMatrix(numVertices, 3, mxSINGLE_CLASS, mxREAL);
    plhs[1] = mxCreateNumericMatrix(numTriangles, 3, mxUINT32_CLASS, mxREAL);
        float*Vertices = (float*mxGetData(plhs[0]);
        unsigned int*Indices = (unsigned int*mxGetData(plhs[1]);

        for (int i = 0; i < 3; ++i)
        {
           memcpy(Vertices + i *numVertices, vertices[i], numVertices
           *sizeof(float));
           free(vertices[i]);
    }
    for (int i = 0; i < 3; ++i)
    {
       memcpy(Indices + i *numTriangles, indices[i], numTriangles
       *sizeof(unsigned int));
       free(indices[i]);
    }
}
```

通常，我们首先检测输入数据类型。调用函数 marchingCubes(...)，并把结果复制到 MATLAB 数组中。

7.5.2 步骤 2

函数 marchingCubes(...) 在头文件 MarchingCubes.h 中声明。创建并保存头文件如下：

```
MarchingCubes.h
#ifndef __MARCHINGCUBES_H__
#define __MARCHINGCUBES_H__

#include <vector>

extern void marchingCubes(float isovalue,
                          float*X, float*Y, float*Z, float*V,
                          int sizeX, int sizeY, int sizeZ,
```

```
                                    float*vertices[],
                                    unsigned int*indices[],
                                    int& numVertices, int& numTriangles);

#endif // __MARCHINGCUBES_H__
```

7.5.3　步骤 3

现在，利用 CUDA 内核和运行时库实现 MarchingCubes.cu 中的 marchingCubes(...)函数，也只给出部分表值，完整表值见 MarchingCubes.cu：

```
MarchingCubes.cu
#include "MarchingCubes.h"

__constant__ unsigned int edgeTable[256] =
{
       0,  265,  515,  778, 1030, 1295, 1541, 1804,
    2060, 2309, 2575, 2822, 3082, 3331, 3593, 3840,
    ...
    ...
    1804, 1541, 1295, 1030,778,515,265,0
};

__constant__ int triTable[256][16] =
{
    { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 },
    {0,8,3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 },
    ...
    ...
    { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 }
};

__device__ float3 interpolatePos(float isovalue, float4 voxel1, float4 voxel2)
{
    float scale = (isovalue - voxel1.w) / (voxel2.w - voxel1.w);
    float3 pos;
    pos.x = voxel1.x + scale *(voxel2.x - voxel1.x);
    pos.y = voxel1.y + scale *(voxel2.y - voxel1.y);
    pos.z = voxel1.z + scale *(voxel2.z - voxel1.z);
    return pos;
}

#define MAX_VERTICES 15
__global__ void getTriangles(float isovalue,
                             float*X, float*Y, float*Z, float*V,
                             int sizeX, int sizeY, int sizeZ,
                             float3*vertexBin, int*triCounter)
{
```

```
int i = blockIdx.x *blockDim.x + threadIdx.x;
// compute capability >= 2.x
//int j = blockIdx.y *blockDim.y + threadIdx.y;
//int k = blockIdx.z *blockDim.z + threadIdx.z;

// compute capability < 2.x
int gy = (sizeY + blockDim.y - 1) / blockDim.y;
int j = (blockIdx.y % gy) *blockDim.y + threadIdx.y;
int k = (blockIdx.y / gy) *blockDim.z + threadIdx.z;

if (i >= sizeX - 1 || j >= sizeY - 1 || k >= sizeZ - 1)
   return;

float4 voxels[8];
float3 isoPos[12];

int idx[8];
idx[0] = sizeX *(sizeY *k + j) + i;
idx[1] = sizeX *(sizeY *k + j + 1) + i;
idx[2] = sizeX *(sizeY *k + j + 1) + i + 1;
idx[3] = sizeX *(sizeY *k + j) + i + 1;
idx[4] = sizeX *(sizeY *(k + 1) + j) + i;
idx[5] = sizeX *(sizeY *(k + 1) + j + 1) + i;
idx[6] = sizeX *(sizeY *(k + 1) + j + 1) + i + 1;
idx[7] = sizeX *(sizeY *(k + 1) + j) + i + 1;

// cube
for (int n = 0; n < 8; ++n)
{
    voxels[n].w = V[idx[n]];
    voxels[n].x = X[idx[n]];
    voxels[n].y = Y[idx[n]];
    voxels[n].z = Z[idx[n]];
}

// find the cube index
unsigned int cubeIndex = 0;
for (int n = 0; n < 8; ++n)
{
    if (voxels[n].w >= isovalue)
    cubeIndex |= (1 << n);
}

// get edges from edgeTable
unsigned int edges = edgeTable[cubeIndex];
if (edges == 0)
   return;

// check 12 edges
if (edges & 1)
```

```
        isoPos[0] = interpolatePos(isovalue, voxels[0], voxels[1]);
    if (edges & 2)
        isoPos[1] = interpolatePos(isovalue, voxels[1], voxels[2]);
    if (edges & 4)
        isoPos[2] = interpolatePos(isovalue, voxels[2], voxels[3]);
    if (edges & 8)
        isoPos[3] = interpolatePos(isovalue, voxels[3], voxels[0]);
    if (edges & 16)
        isoPos[4] = interpolatePos(isovalue, voxels[4], voxels[5]);
    if (edges & 32)
        isoPos[5] = interpolatePos(isovalue, voxels[5], voxels[6]);
    if (edges & 64)
        isoPos[6] = interpolatePos(isovalue, voxels[6], voxels[7]);
    if (edges & 128)
        isoPos[7] = interpolatePos(isovalue, voxels[7], voxels[4]);
    if (edges & 256)
        isoPos[8] = interpolatePos(isovalue, voxels[0], voxels[4]);
    if (edges & 512)
        isoPos[9] = interpolatePos(isovalue, voxels[1], voxels[5]);
    if (edges & 1024)
        isoPos[10] = interpolatePos(isovalue, voxels[2], voxels[6]);
    if (edges & 2048)
        isoPos[11] = interpolatePos(isovalue, voxels[3], voxels[7]);

    // walk through the triTable and get the triangle(s) vertices
    float3 vertices[15];
    int numTriangles = 0;
    int numVertices = 0;

    for (int n = 0; n < 15; n += 3)
    {
        int edgeNumger = triTable[cubeIndex][n];
        if (edgeNumger < 0)
            break;
        vertices[numVertices++] = isoPos[edgeNumger];
        vertices[numVertices++] = isoPos[triTable[cubeIndex][n+1]];
        vertices[numVertices++] = isoPos[triTable[cubeIndex][n+2]];
        ++numTriangles;
    }

    triCounter[idx[0]] = numTriangles;
    for (int n = 0; n < numVertices; ++n)
        vertexBin[MAX_VERTICES *idx[0] + n] = vertices[n];
}

void marchingCubes(float isovalue,
                   float*X, float*Y, float*Z, float*V,
                   int sizeX, int sizeY, int sizeZ,
                   float*vertices[],
                   unsigned int*indices[],
```

```
                    int& numVertices, int& numTriangles)
{
    float*devX = 0;
    float*devY = 0;
    float*devZ = 0;
    float*devV = 0;
    float3*devVertexBin = 0;
    int*devTriCounter = 0;

    int totalSize = sizeX *sizeY *sizeZ;
    cudaMalloc(&devX, sizeof(float) *totalSize);
    cudaMalloc(&devY, sizeof(float) *totalSize);
    cudaMalloc(&devZ, sizeof(float) *totalSize);
    cudaMalloc(&devV, sizeof(float) *totalSize);
    cudaMemcpy(devX, X, sizeof(float) *totalSize, cudaMemcpyHostToDevice);
    cudaMemcpy(devY, Y, sizeof(float) *totalSize, cudaMemcpyHostToDevice);
    cudaMemcpy(devZ, Z, sizeof(float) *totalSize, cudaMemcpyHostToDevice);
    cudaMemcpy(devV, V, sizeof(float) *totalSize, cudaMemcpyHostToDevice);

    cudaMalloc(&devVertexBin, sizeof(float3) * totalSize *MAX_VERTICES);
    cudaMemset(devVertexBin, 0, sizeof(float3) *totalSize *MAX_VERTICES);
    cudaMalloc(&devTriCounter, sizeof(int) *totalSize);
    cudaMemset(devTriCounter, 0, sizeof(int) *totalSize);

    dim3 blockSize(4, 4, 4);

    // compute capability >= 2.x
    //dim3 gridSize((sizeX + blockSize.x - 1) / blockSize.x,
    //               (sizeY + blockSize.y - 1) / blockSize.y,
    //               (sizeZ + blockSize.z - 1) / blockSize.z);

    // compute capabiltiy < 2.x
    int gy = (sizeY + blockSize.y - 1) / blockSize.y;
    int gz = (sizeZ + blockSize.z - 1) / blockSize.z;
    dim3 gridSize((sizeX + blockSize.x - 1) / blockSize.x,
                  gy *gz,
                  1);

    getTriangles<<<gridSize, blockSize>>>(isovalue,
                                          devX, devY, devZ, devV,
                                          sizeX, sizeY, sizeZ,
                                          devVertexBin,
                                          devTriCounter);
    float3*vertexBin = (float3*malloc(sizeof(float3) *totalSize
                         *MAX_VERTICES);
    cudaMemcpy(vertexBin, devVertexBin, sizeof(float3) *totalSize
              *MAX_VERTICES, cudaMemcpyDeviceToHost);
    int*triCounter = (int*malloc(sizeof(int) *totalSize);
    cudaMemcpy(triCounter, devTriCounter, sizeof(int) *totalSize,
              cudaMemcpyDeviceToHost);
```

```
        numTriangles = 0;
    for (int i = 0; i < totalSize; ++i)
        numTriangles + = triCounter[i];
    numVertices = 3 *numTriangles;

    for (int i = 0; i < 3; ++i)
    {
        vertices[i] = (float*malloc(sizeof(float) *numVertices);
        indices[i]= (unsigned int*malloc(sizeof(unsigned int) *numTriangles);
    }
int tIdx = 0, vIdx = 0;
for (int i = 0; i < totalSize; ++i)
{
    int triCount = triCounter[i];
    if (triCount < 1)
        continue;

    int binIdx = i *MAX_VERTICES;
    for (int c = 0; c < triCount; ++c)
    {
            for (int v = 0; v < 3; ++v)
            {
                vertices[0][vIdx] = vertexBin[binIdx].x;
                vertices[1][vIdx] = vertexBin[binIdx].y;
                vertices[2][vIdx] = vertexBin[binIdx].z;
                indices[v][tIdx] = 3 *tIdx + v + 1;
                ++vIdx;
                ++binIdx;
            }
            ++tIdx;
      }
    }

    cudaFree(devX);
    cudaFree(devY);
    cudaFree(devZ);
    cudaFree(devV);
    cudaFree(devVertexBin);
    cudaFree(devTriCounter);
}
```

这一算法的基本架构与采用 MATLAB 直接实现的方法相同。这里看不到嵌套循环，而是给每个体素分配一个线程。在 marchingCubes(...) 函数中，首先利用 CUDA 函数给所有输入输出数据分配 GPU 设备存储器。然后，将所有的输入数据从主机复制到 GPU 设备中。分配一个三维大小为 4×4×4 的线程块。每个线程块提供 64 个线程。一旦定义好块的大小，就可以利用块的大小定义线程网格的大小。根据 GPU 的计算能力，网格线程的大小可以是三维或者二维的。在本范例中，假定

计算能力小于 2.x。然后，CUDA 内核中使用的线程网格大小和线程块大小，用于决定感兴趣体素的位置。

在 marchingCubes(. . .) 函数中，线程网格的 y 和 z 大小以如下方式组合：

```
// compute capability >= 2.x
//dim3 gridSize((sizeX + blockSize.x - 1) / blockSize.x,
//              (sizeY + blockSize.y - 1) / blockSize.y,
//              (sizeZ + blockSize.z - 1) / blockSize.z);

// compute capabiltiy < 2.x
int gy = (sizeY + blockSize.y - 1) / blockSize.y;
int gz = (sizeZ + blockSize.z - 1) / blockSize.z;
dim3 gridSize((sizeX + blockSize.x - 1) / blockSize.x,
              gy *gz,
              1);
```

然后，在 getTriangles kernel(...)中，y 和 z 线程网格大小可以如下方式恢复：

```
int i = blockIdx.x *blockDim.x + threadIdx.x;

// compute capability >= 2.x
//int j = blockIdx.y *blockDim.y + threadIdx.y;
//int k = blockIdx.z *blockDim.z + threadIdx.z;

// compute capability < 2.x
int gy = (sizeY + blockDim.y - 1) / blockDim.y;
int j = (blockIdx.y % gy) *blockDim.y + threadIdx.y;
int k = (blockIdx.y / gy) *blockDim.z + threadIdx.z;
```

一旦利用线程和线程块索引确定体素的位置，剩余部分与 MATLAB 方法相同。

7.5.4 步骤 4

代码准备好后，编译生成如下 c-mex 文件：

```
buildSurfaceCuda.m
system('nvcc -c MarchingCubes.cu -Xptxas -v');
mex getSurfaceCuda.cpp MarchingCubes.obj -lcudart -L"C:\Program Files
\NVIDIA GPU Computing Toolkit\CUDA\v5.0\lib\x64" -I"C:\Program Files
\NVIDIA GPU Computing Toolkit\CUDA\v5.0\include" —v
```

在 nvcc 中，添加 −Xptxas 选项。这个选项可以打印出附加信息，这些附加信息有利于判定内核消耗了多少存储器。当有来自 CUDA 的关于存储器资源的错误信息时，这个操作就会变得得心应手。下面是 −Xptxas 的打印输出样例：

```
ptxas : info : 0 bytes gmem, 17408 bytes cmem[0]
ptxas : info : Compiling entry function '_Z12getTrianglesfPfS_S_S_
iiiP6float3Pi' for 'sm_10'
ptxas : info : Used 38 registers, 88 bytes smem, 68 bytes cmem[1], 324 bytes
lmem
```

运行 MATLAB 代码后，将得到 c-mex 文件，这个文件可以在 MATLAB 命令窗口中调用。

7.5.5 步骤 5

现在，修改并运行测试代码如下：

```
% generate sample volume data
[X, Y, Z, V] = flow;
X = single(X);
Y = single(Y);
Z = single(Z);
V = single(V);
isovalue = single(-3);
[Vertices3, Indices3] = getSurfaceCuda(X, Y, Z, V, isovalue);

% visualize triangles
figure
p = patch('Faces', Indices3, 'Vertices', Vertices3);
set(p, 'FaceColor', 'none', 'EdgeColor', 'blue');
daspect([1,1,1])
view(3);
camlight
lighting gouraud
grid on
```

利用 GPU 计算，生成新的三角形图像，如图 7.13 所示。

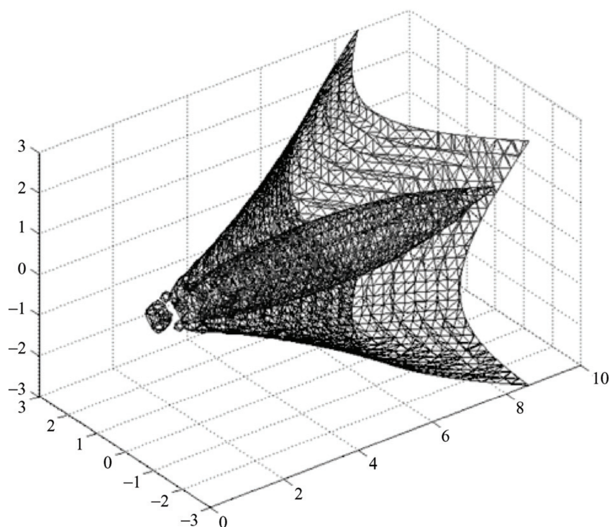
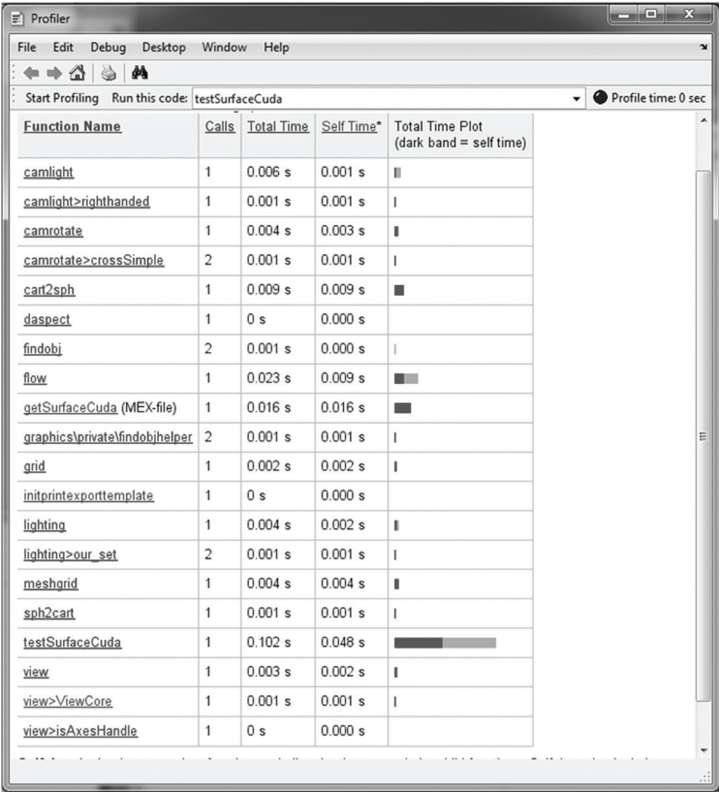


图 7.13 利用 c-mex 和 GPU，等值面值=−3 时的等值面

7.5.6 时间分析

进行时间分析，看看利用 GPU 性能改善了多少。在 Profile 中运行 testSurfaceCuda.m，如图 7.14 所示。



Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
camlight	1	0.006 s	0.001 s	
camlight>righthanded	1	0.001 s	0.001 s	
camrotate	1	0.004 s	0.003 s	
camrotate>crossSimple	2	0.001 s	0.001 s	
cart2sph	1	0.009 s	0.009 s	
daspect	1	0 s	0.000 s	
findobj	2	0.001 s	0.000 s	
flow	1	0.023 s	0.009 s	
getSurfaceCuda (MEX-file)	1	0.016 s	0.016 s	
graphics\private\findobjhelper	2	0.001 s	0.001 s	
grid	1	0.002 s	0.002 s	
initprintexporttemplate	1	0 s	0.000 s	
lighting	1	0.004 s	0.002 s	
lighting>our_set	2	0.001 s	0.001 s	
meshgrid	1	0.004 s	0.004 s	
sph2cart	1	0.001 s	0.001 s	
testSurfaceCuda	1	0.102 s	0.048 s	
view	1	0.003 s	0.002 s	
view>ViewCore	1	0.001 s	0.001 s	
view>isAxesHandle	1	0 s	0.000 s	

图 7.14 利用 GPU 的 c-mex 时间分析

现在利用 GPU，0.016s 内可以完成 getSurfaceCuda(...) 中所有的三角形的计算，这一速度大约是第二种实现方法的两倍。

7.6 总结

本章探索了 Marching Cubes 算法的三种实现方法。

第一种是直接实现，不进行任何优化。

第二种实现方法，利用向量化和预分配的概念提高了速度。在这一实现方法中，去掉了三重嵌套循环，速度提升相当可观。但是，当体素增加时，要注意可能会遇到存储器资源不足的问题。

第三种方法，先将 GPU 引入到第一种实现方法中。利用 CUDA 内核替换所

有内层循环操作，分配 GPU 的线程来计算每一个体素的三角形。然后把所有的输出数据复制回 MATLAB 数组中。

总体来说，每一种优化技术都可以获得速度的提升，如下表所述：

方法	时间/s
getSurfaceNoOpt	1.9
getSurfaceWithOpt	0.029
getSurfaceCuda	0.016

第 8 章　CUDA 转换实例：3D 图像处理

8.1　本章学习目标

三维（3D）医学图像处理包含海量数据，属于密集计算领域的一种。通过 3D 医学图像处理实例，我们可以体验从 MATLAB 到 CUDA 的变换流程，而且在性能方面感受到实际处理速度的大幅提升。本章将讨论以下问题：

- c-mex 代码的 CUDA 转换实例。
- 耗时结果分析与 CUDA 转换设计。
- 大文件的实际 CUDA 转换。

8.2　基于 Atlas 分割方法的 MATLAB 代码

8.2.1　基于 Atlas 分割背景知识

在各种医疗图像临床和研究中，从患者的 3D 图像数据中精确地圈定目标解剖是大有益处的。在大规模患者研究中，采用人工方式处理如此大量的数据极为耗时和乏味，而且受限于不同观察者之间的差异，因此对目标器官图像进行可靠的自动分割就非常关键。从 3D 医疗图像中进行器官图像自动分割是当前医疗图像处理中的热点问题。

基于 Atlas 的分割方法广泛应用于医疗图像分析。基于 Atlas 分割方法采用图像配准技术传播 Atlas 图像的分割。一个 Atlas 数据集由两个图像组成：患者强度图像和对应的二进制分割模板，二进制分割模板由手工产生，如图 8.1 所示。

图 8.2 给出了单一基于 Atlas 分割方法的概念。当拿到一个需要进行分割的图像数据时，将 Atlas 强度图像与新的目标图像进行配准。通过图像配准，能够获得从 Atlas 强度图像到新的目标图像映射的 T 变换。通过对 Atlas 分割模板进行 T 变换，为新的目标图像产生一个新的分割模板。

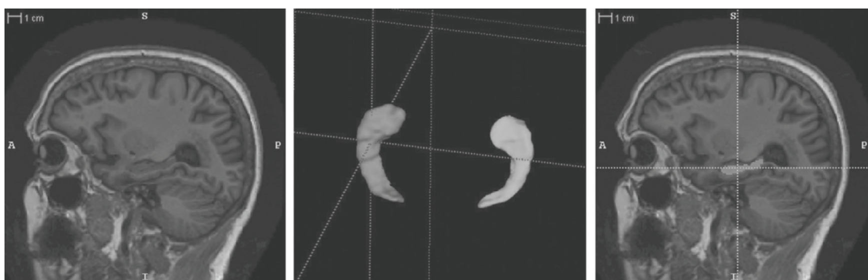


图 8.1　Atlas 集：患者 MRI 图像（左）、对应的分割模板（中）、重叠图像（右）

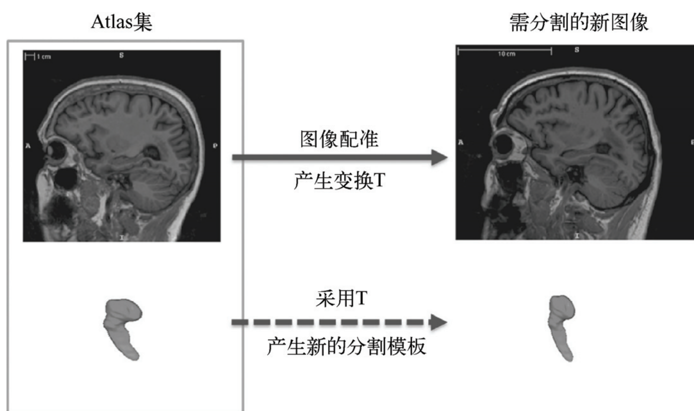


图 8.2　单一基于 atlas 分割背后的原理

基于 atlas 的分割方法具有很多优点，即使目标对象具有非常模糊和复杂的边界，该方法仍可以得到分割结果。基于 atlas 的分割方法不足之处在于，由于分割过程中需要采用 3D 图像配准，因此需要较大的计算能力。由于 3D 图像配准是一个计算开销很大的迭代处理，因此当使用原始大小的医疗图像时，运行非常缓慢。

8.2.2　用于分割的 MATLAB 代码

让我们从纯 m 代码开始进行基于 atlas 的海马体分割。在 m_code_Only 示例文件夹，主模块 atlasSeg_Main.m 首先载入数据文件 medical3D_data.mat，该文件包括一个 atlas 集（atlasVol, atlasSegVol）和目标图像（targetVol）。图 8.3 和图 8.4 给出了 atlasVol 数据和 targetVol 数据的全部强度图像。可以通过改变 atlasSeg_Main.m 第 6 行中 viewBx、viewBy 和 viewBz 的值调整显示平面。

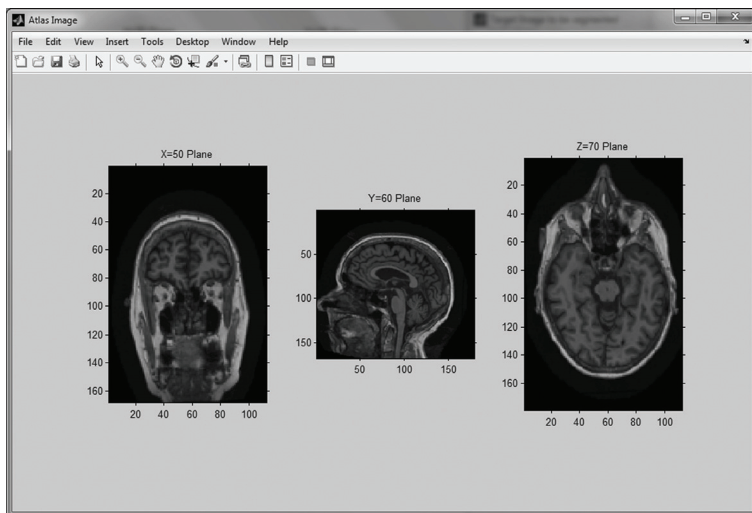


图 8.3　*X*=50、*Y*=60 和 *Z*=70 平面上的 3D atlas 强度图像（atlasVol）

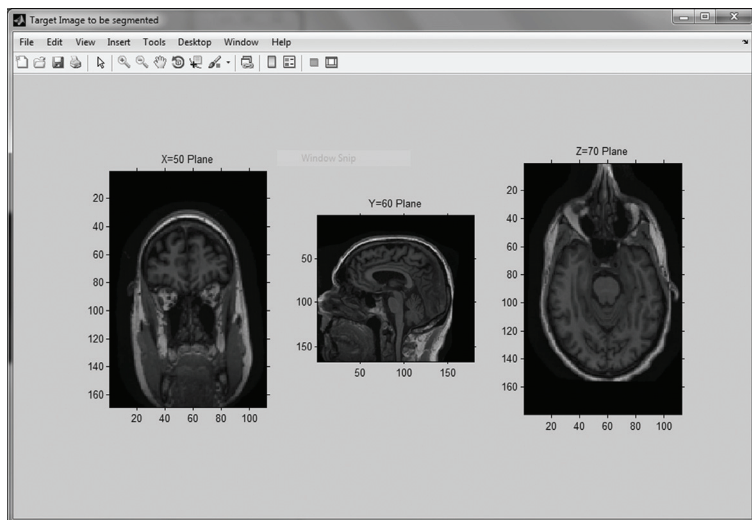


图 8.4　*X*=50、*Y*=60 和 *Z*=70 平面上的 3D 目标强度图像（targetVol）

```
% atlasSeg_Main.m
% Single-Atlas-Based Hippocampus Segmentation

clear all;
close all;
load('medical3D_data');

viewBx = 50; viewBy = 60; viewBz = 70;
```

```
myView2('Atlas Image',atlatVol, viewBx, viewBy, viewBz);
myView2('Target Image to be segmented',targetVol,viewBx,viewBy,
  viewBz);

viewLHippoX = 30; viewLHippoY = 28; viewLHippoZ = 25;
viewRHippoX = 28; viewRHippoY = 24; viewRHippoZ = 23;

% cropping margin around manual segmentation in atlas image
margin = 20;
iterLimit = 1000;

% objNum = 1 for left, objNum = 2 for right
tic

%
% For Left Hippocampus
%
objNum = 1;
[cropMovVol_Left, minPosX, maxPosX, minPosY, maxPosY, minPosZ,
  maxPosZ ] = ...
crop_around_ROI(atlatVol, atlasSegVol, objNum, margin);
cropTargetVol_Left = targetVol(minPosX - margin:maxPosX + margin,
  ...
minPosY - margin:maxPosY + margin, ...
minPosZ - margin:maxPosZ + margin);
myView2('Target Image around Left ROI',cropTargetVol_Left,
  viewLHippoX, viewLHippoY, viewLHippoZ);

movSeg_Left = atlasSegVol(minPosX - margin:maxPosX + margin, ...
minPosY - margin:maxPosY + margin, ...
minPosZ - margin:maxPosZ + margin);

% if the cropped region may include right segment by margin, we clean
  it up here.
movSeg_Left = double(movSeg_Left = = 1);
myView_overlap_red('Atlas Set (Moving Intensity Image and its
  manuaal segmentation) around left ROI',...
cropMovVol_Left, viewLHippoX,viewLHippoY,viewLHippoZ,
movSeg_Left);

% Make similar range of intensities for both 3D image for better
registration
movVol_org = cropMovVol_Left;
refVol_org = cropTargetVol_Left;
[adjMov, adjRef, adjMovRatio, maxOrgMov, minOrgMov] = rough_int_
```

```
adjust3(movVol_org, refVol_org);

movVol = adjMov;
refVol = adjRef;

% 3D deformable registration
[movVol_updated, Tx, Ty, Tz] = deformableRegister3D(movVol,
refVol, iterLimit);

intAdjustBackMov = rough_int_adjustBack(movVol_updated,
adjMovRatio, maxOrgMov, minOrgMov);
movSegVol_updatedNN = interp3(movSeg_Left, Ty, Tx, Tz, 'nearest');

myView_overlap_red('Registration Result from Left ROI of Atlas
set',...
intAdjustBackMov, viewLHippoX,viewLHippoY,viewLHippoZ,
movSegVol_updatedNN);
myView_overlap_red('Segmentation Result for Left ROI of Target
Image',...
cropTargetVol_Left, viewLHippoX,viewLHippoY,viewLHippoZ,
movSegVol_updatedNN);

%
% For Right Hippocampus
%
objNum = 2;
[cropMovVol_Right, minPosX, maxPosX, minPosY, maxPosY, minPosZ,
maxPosZ ] = ...
crop_around_ROI(atlatVol, atlasSegVol, objNum, margin);

cropTargetVol_Right = targetVol(minPosX - margin:maxPosX + margin,
...
minPosY - margin:maxPosY + margin, ...
minPosZ - margin:maxPosZ + margin);
myView2('Target Image around Right ROI',cropTargetVol_Right,
viewRHippoX, viewRHippoY, viewRHippoZ);

movSeg_Right = atlasSegVol(minPosX - margin:maxPosX + margin, ...
minPosY - margin:maxPosY + margin, ...
minPosZ - margin:maxPosZ + margin);
% if the cropped region may include right segment by margin, we clean
it up here.

movSeg_Right = double(movSeg_Right = = 2);
```

```
62 myView_overlap_green('Atlas Set (Moving Intensity Image and its
   manuaal segmentation) around Right ROI',...
63 cropMovVol_Right, viewRHippoX,viewRHippoY,viewRHippoZ,
   movSeg_Right);

64 % Make similar range of intensities for both 3D image for better
   registration
65 movVol_org = cropMovVol_Right;
66 refVol_org = cropTargetVol_Right;
67 [adjMov, adjRef, adjMovRatio, maxOrgMov, minOrgMov] = ...
68 rough_int_adjust3(movVol_org, refVol_org);

69 movVol = adjMov;
70 refVol = adjRef;

71 % 3D deformable registration
72 [movVol_updated, Tx, Ty, Tz] = deformableRegister3D(movVol, refVol,
   iterLimit);

73 intAdjustBackMov = rough_int_adjustBack(movVol_updated,
   adjMovRatio, maxOrgMov, minOrgMov);
74 movSegVol_updatedNN = interp3(movSeg_Right, Ty, Tx, Tz, 'nearest');

75 myView_overlap_green('Registration Result from Right ROI of Atlas
   set',...
76 intAdjustBackMov, viewRHippoX,viewRHippoY,viewRHippoZ,
   movSegVol_updatedNN);
77 myView_overlap_green('Segmentation Result for Right ROI of Target
   Image',...
78 cropTargetVol_Right, viewRHippoX,viewRHippoY,viewRHippoZ,
   movSegVol_updatedNN);
79 toc
```

为了减小配准过程中的计算量，需裁剪图像至关注区域（Region of Interest，ROI）（见 atlasSeg_Main.m 中 20～24 行）。因为大脑包括两个海马体，我们将 atlas 图像和目标图像裁剪为左 ROI 和右 ROI 两个区域。基于 atlas 手动分割目标进行区域裁剪，并适当留有余量。

图 8.5 给出了从围绕左海马体分割模板（在 X、Y 和 Z 三个方向各留出了 20 个像素的余量）的 atlas 图像得到的裁剪结果。图 8.5 中的第一行给出了 ROI 的强度 atlas 图像，第二行给出了对应强度 atlas 图像的手动分割图像，第三行给出了重叠图像。左海马体模板值设为 1（见第 19 行中 objNum=1），而右海马体模板值设为 2（见第 50 行中 objNum=2）。这样基于模板值，就可以很容易地将它们分为右

ROI 和左 ROI。图 8.6 给出了围绕右海马体分割模板的裁剪结果。

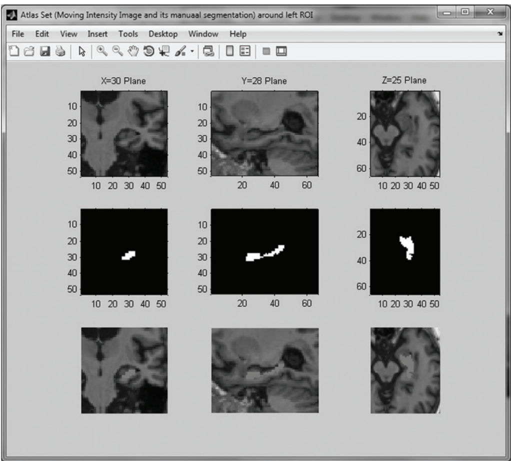


图 8.5 *X*=30、*Y*=28 和 *Z*=25 平面上 atlas 图像集的左 ROI（cropMovVol_Left），对于左海马体，模板值设为 1（如最下一排图像所示）

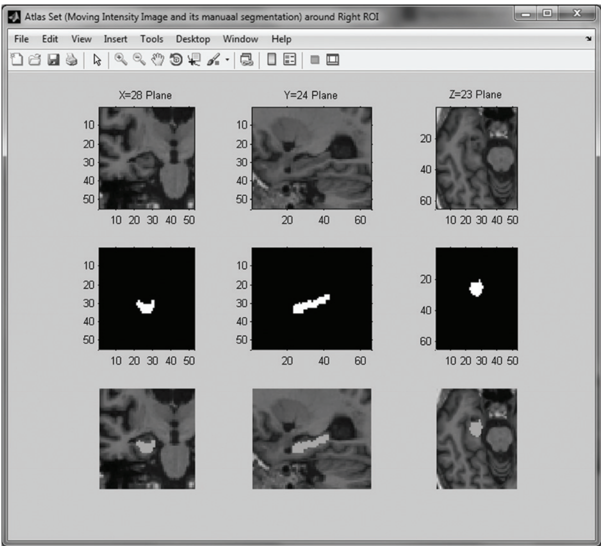


图 8.6 *X*=28、*Y*=24 和 *Z*=23 平面上 atlas 图像集的右 ROI（cropMovVol_Right），对于右海马体，模板值设为 2（如最下一排图像所示）

图 8.7 和图 8.8 给出了目标图像的裁剪结果，且目标图像和切割 atlas 图像有相同的坐标范围。

由于要将裁剪 atlas 图像转换为裁剪目标图像，因此称裁剪 atlas 图像为移动图像（Moving Image，见第 34 行），裁剪目标图像为参考图像（Reference Image，见

第 35 行）。在进行移动图像和参考图像类似强度范围的简单配准过程后（见第 36 行的 rough_int_adjust3），运行第 40 行的 deformableRegister3D。应用 3D 转换（T_x，T_y，T_z）将 atlas 分割模板（见第 42 行）转换为目标图像的最终分割模板。结果如图 8.9 和图 8.10 所示。能够通过设置左海马体的 viewLHippoX、viewLHippoY、viewLHippoZ 和右海马体的 viewRHippoX、viewRHippoY、viewRHippoZ 调整结果平面。



图 8.7　X=30、Y=28 和 Z=25 平面上目标图像集的左 ROI（cropTargetVol_Left）

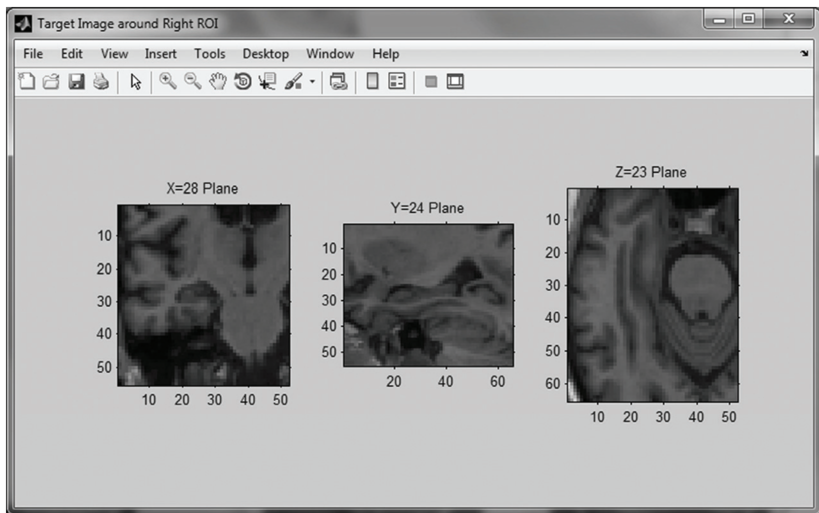


图 8.8　X=28、Y=24 和 Z=23 平面上目标图像集的右 ROI（cropTargetVol_Right）

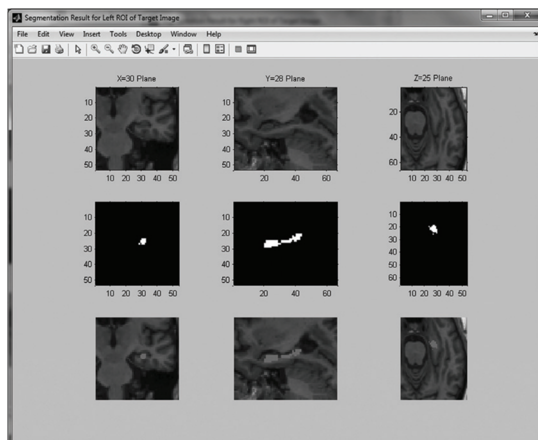


图 8.9　左海马体的目标分割结果

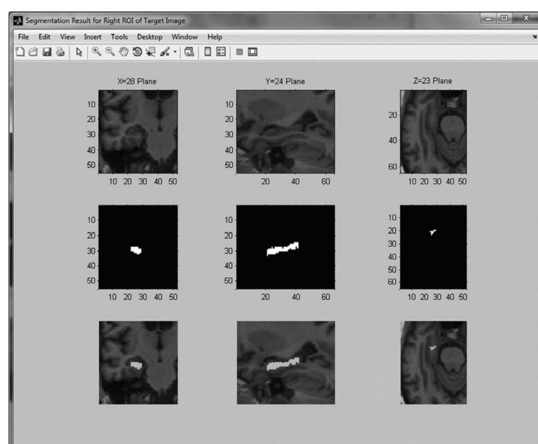


图 8.10　右海马体的目标分割结果

```
iter =

   998


iter =

   999


iter =

        1000

Elapsed time is 609.036528 seconds.
```

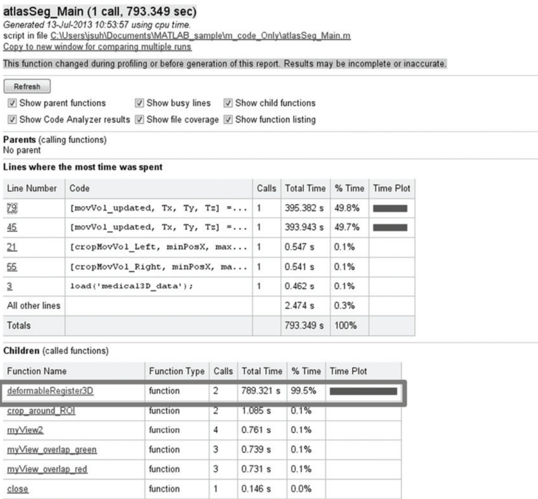
图 8.11　atlasSeg_Main.m 运行时间

尽管图 8.9 和图 8.10 中的分割结果看起来很合适，但是左海马体和右海马体的 atlasSeg_Main.m 运行时间均约为 609 s。下面分析整个运算过程最为耗时的运算子模块。

8.3　通过分析进行 CUDA 最优设计

8.3.1　分析 MATLAB 代码

现在，在 MATLAB 中使用 Profiler 分析代码。输入主文件名 atlasSeg_Main 并运行。图 8.12 为分析结果，从图中能够清晰地看到，总运行时间的 95%来自函数 deformableRegiser3D。如果在分析结果中点击 deformableRegiser3D 行，能够获得如图 8.13 所示更详细的分析结果。



atlasSeg_Main (1 call, 793.349 sec)
Generated 13-Jul-2013 10:53:57 using cpu time.
script in file C:\Users\jsuh\Documents\MATLAB_sample\m_code_Only\atlasSeg_Main.m
Copy to new window for comparing multiple runs

This function changed during profiling or before generation of this report. Results may be incomplete or inaccurate.

Refresh

☑ Show parent functions ☑ Show busy lines ☑ Show child functions
☑ Show Code Analyzer results ☑ Show file coverage ☑ Show function listing

Parents (calling functions)
No parent

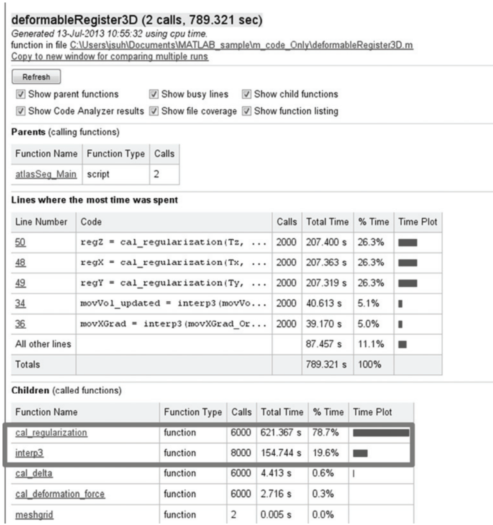
Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
79	[movVol_updated, Tx, Ty, Tz] =...	1	395.382 s	49.8%	
45	[movVol_updated, Tx, Ty, Tz] =...	1	393.943 s	49.7%	
21	[cropMovVol_Left, minPosX, max...	1	0.547 s	0.1%	
55	[cropMovVol_Right, minPosX, ma...	1	0.541 s	0.1%	
3	load('medical3D_data');	1	0.462 s	0.1%	
All other lines			2.474 s	0.3%	
Totals			793.349 s	100%	

Children (called functions)

Function Name	Function Type	Calls	Total Time	% Time	Time Plot
deformableRegister3D	function	2	789.321 s	99.5%	
crop_around_ROI	function	2	1.085 s	0.1%	
myView2	function	4	0.761 s	0.1%	
myView_overlap_green	function	3	0.739 s	0.1%	
myView_overlap_red	function	3	0.731 s	0.1%	
close	function	1	0.146 s	0.0%	

图 8.12　主函数分析结果（atlasSeg_Main.m）



deformableRegister3D (2 calls, 789.321 sec)
Generated 13-Jul-2013 10:55:32 using cpu time.
function in file C:\Users\jsuh\Documents\MATLAB_sample\m_code_Only\deformableRegister3D.m
Copy to new window for comparing multiple runs

Refresh

☑ Show parent functions ☑ Show busy lines ☑ Show child functions
☑ Show Code Analyzer results ☑ Show file coverage ☑ Show function listing

Parents (calling functions)

Function Name	Function Type	Calls
atlasSeg_Main	script	2

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
50	regZ = cal_regularization(Tz, ...	2000	207.400 s	26.3%	
48	regX = cal_regularization(Tx, ...	2000	207.363 s	26.3%	
49	regY = cal_regularization(Ty, ...	2000	207.319 s	26.3%	
34	movVol_updated = interp3(movVo...	2000	40.613 s	5.1%	
36	movXGrad = interp3(movXGrad_Or...	2000	39.170 s	5.0%	
All other lines			87.457 s	11.1%	
Totals			789.321 s	100%	

Children (called functions)

Function Name	Function Type	Calls	Total Time	% Time	Time Plot
cal_regularization	function	6000	621.367 s	78.7%	
interp3	function	8000	154.744 s	19.6%	
cal_delta	function	6000	4.413 s	0.6%	
cal_deformation_force	function	6000	2.716 s	0.3%	
meshgrid	function	2	0.005 s	0.0%	

图 8.13　消耗大部分主函数运算时间的 deformableRegister3D 模块分析结果

```
function [movVol_updated, Tx, Ty, Tz] = deformableRegister3D(movVol, refVol, iterLimit)
[refX, refY, refZ] = size(refVol)
[Ty,Tx,Tz] = meshgrid(1:refY,1:refX,1:refZ);

% calculate gradient for moving volume
[movX, movY, movZ] = size(movVol)
movXGrad_Org = zeros(movX, movY, movZ);
movYGrad_Org = zeros(movX, movY, movZ);
movZGrad_Org = zeros(movX, movY, movZ);

movXGrad = zeros(movX, movY, movZ);
movYGrad = zeros(movX, movY, movZ);
movZGrad = zeros(movX, movY, movZ);
movXGrad_Org(2:end-1,2:end-1,2:end-1) = movVol(3:end,2:end-1,2:end-1) - movVol(1:end-2,2:end-1,2:end-1);
movYGrad_Org(2:end-1,2:end-1,2:end-1) = movVol(2:end-1,3:end,2:end-1) - movVol(2:end-1,1:end-2,2:end-1);
movZGrad_Org(2:end-1,2:end-1,2:end-1)=movVol(2:end-1,2:end-1,3:end) - movVol(2:end-1,2:end-1,1:end-2);

forceX = zeros(refX, refY, refZ);
forceY = zeros(refX, refY, refZ);
forceZ = zeros(refX, refY, refZ);

regX = zeros(refX, refY, refZ);
regY = zeros(refX, refY, refZ);
regZ = zeros(refX, refY, refZ);

delta_Tx = zeros(refX, refY, refZ);

delta_Ty = zeros(refX, refY, refZ);
delta_Tz = zeros(refX, refY, refZ);

for iter = 1:iterLimit

iter
movVol_updated = interp3(movVol, Ty, Tx, Tz); % make movVol in the ref space
movXGrad = interp3(movXGrad_Org, Ty, Tx, Tz);
movYGrad = interp3(movYGrad_Org, Ty, Tx, Tz);
movZGrad = interp3(movZGrad_Org, Ty, Tx, Tz);
% calculate deformation force
forceX = cal_deformation_force(refVol, movVol_updated, movXGrad);
forceY = cal_deformation_force(refVol, movVol_updated, movYGrad);
```

```
forceZ = cal_deformation_force(refVol, movVol_updated, movZGrad);
% Regularization
% Regularization makes the registration smoother warps.
regulFactor = 0.1;
regX = cal_regularization(Tx, regulFactor);
regY = cal_regularization(Ty, regulFactor);
regZ = cal_regularization(Tz, regulFactor);

stepSize = 1.0;
delta_Tx = cal_delta(forceX, regX, stepSize);
delta_Ty = cal_delta(forceY, regY, stepSize);
delta_Tz = cal_delta(forceZ, regZ, stepSize);
Tx = Tx + delta_Tx;
Ty = Ty + delta_Ty;
Tz = Tz + delta_Tz;
Tx = max(min(Tx, refX), 1);
Ty = max(min(Ty, refY), 1);
Tz = max(min(Tz, refZ), 1);
end
```

在 deformableRegister3D 模块的详细分析结果中能够看出 cal_regularization 是最忙的子模块，interp3 是次忙的子模块。图 8.14 标出了运算繁忙模块，即 deformable Register3D 中阴影部分。

仔细研究 cal_regularization 子模块。点击分析结果中的 al_regularization 行，能够看到更详细的分析结果，如图 8.15 和图 8.16 所示。由分析结果可知，大部分时间用于提取周围梯度值（前，后，上，下，左，右）与此处梯度值之差，得到正则化变形。

由图 8.13 可以得出 interp3 是次耗时子模块。点击图 8.13 中的 interp3，得到如图 8.17 所示分析界面。由于在 deformableRegister3D 模块中使用的 interp3 是 MATLAB 内置函数，明确此代码的运算瓶颈不太容易，因为 MATLAB 内置函数通常包含多种选项多层次且是高度优化的。

```
              2   20 forceY = zeros(refX, refY, refZ);
< 0.01        2   21 forceZ = zeros(refX, refY, refZ);
                  22
              2   23 regX = zeros(refX, refY, refZ);
< 0.01        2   24 regY = zeros(refX, refY, refZ);
< 0.01        2   25 regZ = zeros(refX, refY, refZ);
                  26
              2   27 delta_Tx = zeros(refX, refY, refZ);
< 0.01        2   28 delta_Ty = zeros(refX, refY, refZ);
< 0.01        2   29 delta_Tz = zeros(refX, refY, refZ);
                  30
              2   31 for iter = 1:iterLimit
                  32
  0.17     2000   33     iter
 40.61     2000   34     movVol_updated = interp3(movVol, Ty, Tx, Tz);  % make movVol in the ref space
                  35
 39.17     2000   36     movXGrad = interp3(movXGrad_Org, Ty, Tx, Tz);   %f_r => dr_img2
 37.95     2000   37     movYGrad = interp3(movYGrad_Org, Ty, Tx, Tz);
 37.79     2000   38     movZGrad = interp3(movZGrad_Org, Ty, Tx, Tz);
                  39
                  40     % calculate deformation force
  1.22     2000   41     forceX = cal_deformation_force(refVol, movVol_updated, movXGrad);
  1.18     2000   42     forceY = cal_deformation_force(refVol, movVol_updated, movYGrad);
  1.01     2000   43     forceZ = cal_deformation_force(refVol, movVol_updated, movZGrad);
                  44
                  45     % Regularization
                  46     % Regularization makes the registration smoother warps.
< 0.01     2000   47     regulFactor = 0.1;
207.36     2000   48      regX = cal_regularization(Tx, regulFactor);
207.32     2000   49      regY = cal_regularization(Ty, regulFactor);
207.40     2000   50      regZ = cal_regularization(Tz, regulFactor);
                  51
                  52 %     stepSize = 0.5;
           2000   53     stepSize = 1.0;
  1.95     2000   54     delta_Tx = cal_delta(forceX, regX, stepSize);
  1.80     2000   55     delta_Ty = cal_delta(forceY, regY, stepSize);
  1.80     2000   56     delta_Tz = cal_delta(forceZ, regZ, stepSize);
                  57
  0.44     2000   58     Tx = Tx + delta_Tx;
  0.52     2000   59     Ty = Ty + delta_Ty;
  0.49     2000   60     Tz = Tz + delta_Tz;
                  61
```

图 8.14　deformableRegister3D 分析结果，高亮占用最多运算时间的子模块以高亮显示

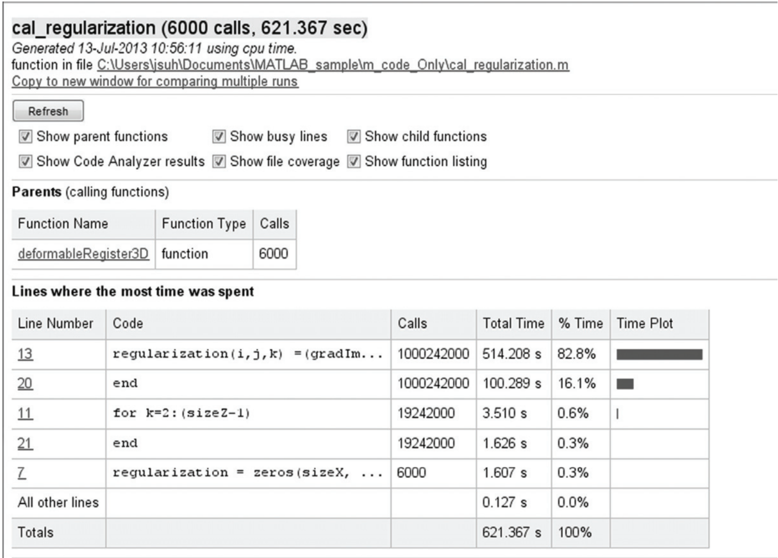


图 8.15　cal_regularization 子模块分析总结

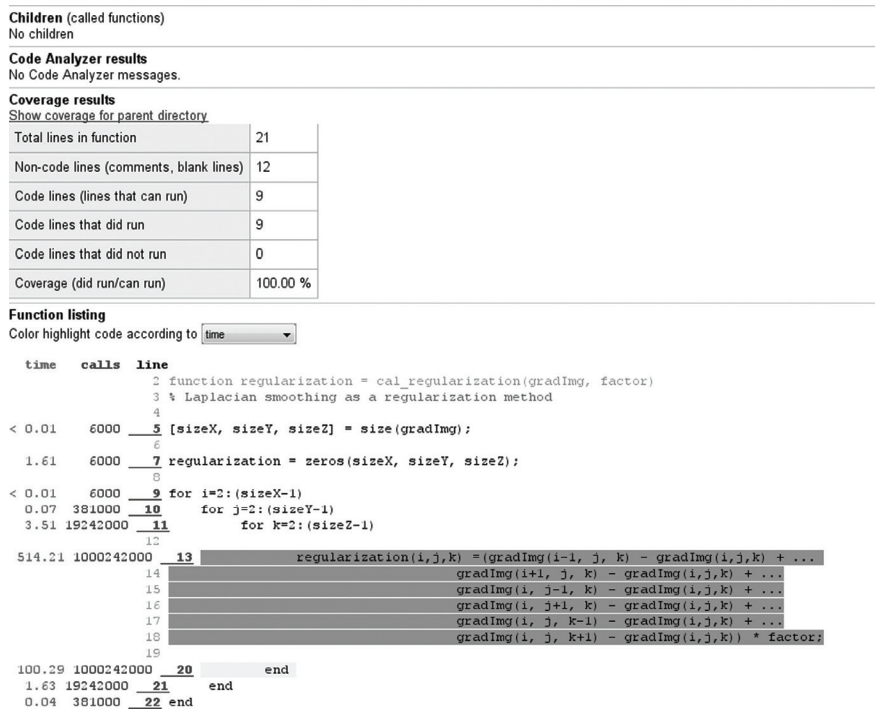
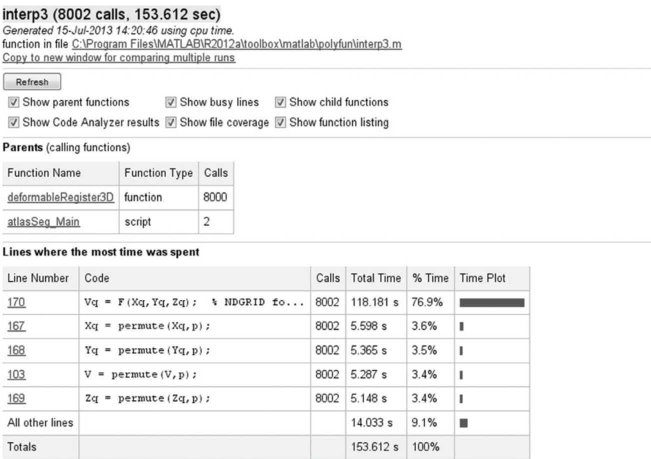


图 8.16　cal_regularization 子模块逐行详细分析



interp3 (8002 calls, 153.612 sec)
Generated 15-Jul-2013 14:20:46 using cpu time.
function in file C:\Program Files\MATLAB\R2012a\toolbox\matlab\polyfun\interp3.m
Copy to new window for comparing multiple runs

Refresh

☑ Show parent functions ☑ Show busy lines ☑ Show child functions
☑ Show Code Analyzer results ☑ Show file coverage ☑ Show function listing

Parents (calling functions)

Function Name	Function Type	Calls
deformableRegister3D	function	8000
atlasSeg_Main	script	2

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
170	Vq = F(Xq,Yq,Zq); % NDGRID fo...	8002	118.181 s	76.9%	
167	Xq = permute(Xq,p);	8002	5.598 s	3.6%	
168	Yq = permute(Yq,p);	8002	5.365 s	3.5%	
103	V = permute(V,p);	8002	5.287 s	3.4%	
169	Zq = permute(Zq,p);	8002	5.148 s	3.4%	
All other lines			14.033 s	9.1%	
Totals			153.612 s	100%	

图 8.17　interp3 子模块分析总结

8.3.2　结果分析和 CUDA 最优设计

从分析结果中能够发现 cal_regularization 是最忙子模块且 interp3 是次忙子模块。所以必须要用 CUDA 和 c-mex 代替这两个模块来提高整个分割进程的速度。但是，再仔细观察 deformableRegister3D 模块。在如下 deformableRegister3D 代码中，cal_regularization 和 interp3 位于迭代循环中，并且有一些其他模块位于其前后。

```
function [movVol_updated, Tx, Ty, Tz] = deformableRegister3D
(movVol, refVol, iterLimit)

  .
  .
  .
for iter = 1:iterLimit

iter
movVol_updated = interp3(movVol, Ty, Tx, Tz); % make movVol in the ref
space
movXGrad = interp3(movXGrad_Org, Ty, Tx, Tz);
movYGrad = interp3(movYGrad_Org, Ty, Tx, Tz);
movZGrad = interp3(movZGrad_Org, Ty, Tx, Tz);
% calculate deformation force
forceX = cal_deformation_force(refVol, movVol_updated,
movXGrad);
forceY = cal_deformation_force(refVol, movVol_updated,
movYGrad);
forceZ = cal_deformation_force(refVol, movVol_updated,
movZGrad);
```

```
% Regularization
% Regularization makes the registration smoother warps.
regulFactor = 0.1;
regX = cal_regularization(Tx, regulFactor);
regY = cal_regularization(Ty, regulFactor);
regZ = cal_regularization(Tz, regulFactor);

stepSize = 1.0;
delta_Tx = cal_delta(forceX, regX, stepSize);
delta_Ty = cal_delta(forceY, regY, stepSize);
delta_Tz = cal_delta(forceZ, regZ, stepSize);
Tx = Tx + delta_Tx;
Ty = Ty + delta_Ty;
Tz = Tz + delta_Tz;
Tx = max(min(Tx, refX), 1);
Ty = max(min(Ty, refY), 1);
Tz = max(min(Tz, refZ), 1);
end
```

所以，如果仅将这两个函数用 CUDA 可用的 c-mex 函数替代，那么迭代中 CPU 和 GPU 内存之间的数据传输在这两个模块前后都需要进行。这些过程中的数据传输需要很多开销并使整个过程效率降低。

因此，为减小 CPU 和 GPU 存储器之间数据传输，最好的办法是将迭代循环全部转换为 GPU 代码，这样能够避免迭代中 CPU 和 GPU 设备之间的全部数据传输。

8.4 CUDA 转换 1——正则化

接下来开始转换 cal_regularization，该子模块是分析显示中最耗时函数。在 m_code_partialCuda 示例文件夹中，主模块 atlasSeg_PartialCuda.m 和之前的 atlasSeg_Main.m 有几乎相同的结构。

```
#include "mex.h"
#include <cuda_runtime.h>
#include "cal_regularization_cuda.h"

// function regularization = cal_regularization(gradImg, factor);
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    if (nrhs != 2)
        mexErrMsgTxt("Invalid number of input arguments");

    if (nlhs != 1)
```

```
        mexErrMsgTxt("Invalid number of outputs");

    for (int i = 0; i < nrhs; ++i)
    {
        if (!mxIsDouble(prhs[i]))
        {
            mexErrMsgTxt("input vector data type must be double");
            break;
        }
    }

    const mwSize* size = mxGetDimensions(prhs[0]);
    int sizeX = size[0];
    int sizeY = size[1];
    int sizeZ = size[2];

    // inputs
    double* gradImg = (double*)mxGetData(prhs[0]);
    double factor = *((double*)mxGetData(prhs[1]));

    // outputs
    plhs[0] = mxCreateNumericArray(3, size, mxDOUBLE_CLASS, mxREAL);
    double* regularization = (double*)mxGetData(plhs[0]);

    // calRegularization cuda
    calRegularization(gradImg, factor, regularization, sizeX, sizeY,
    sizeZ);

    cudaError_t error = cudaGetLastError();
    if (error != cudaSuccess)
    {
        mexPrintf("%s\n", cudaGetErrorString(error));
        mexErrMsgTxt("CUDA failed\n");
    }
}
```

cal_regularization_cuda.h

```
#ifndef __CALREGULARIZATION_H__
#define __CALREGULARIZATION_H__

extern void calRegularization(double* in, double factor, double* out,
                              int sx, int sy, int sz);

#endif // __CALREGULARIZATION_H__
```

calRegularization.cu

```
__global__ void calRegularizationKernel(double* in, double factor,
                                        double* out,
                                        int sx, int sy, int sz)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    // compute capability >= 2.x
    //int j = blockIdx.y * blockDim.y + threadIdx.y;
    //int k = blockIdx.z * blockDim.z + threadIdx.z;
    // compute capability < 2.x
    int gy = (sy + blockDim.y - 1) / blockDim.y;
    int j = (blockIdx.y % gy) * blockDim.y + threadIdx.y;
    int k = (blockIdx.y / gy) * blockDim.z + threadIdx.z;

    if (i < 1 || i >= sx - 1 ||
        j < 1 || j >= sy - 1 ||
        k < 1 || k >= sz - 1)
        return;

    int idx = sx * (sy * k + j) + i;
    double org = in[idx];
    out[idx] = (in[sx * (sy * k + j) + i - 1] - org +
                in[sx * (sy * k + j) + i + 1] - org +
                in[sx * (sy * k + j - 1) + i] - org +
                in[sx * (sy * k + j + 1) + i] - org +
                in[sx * (sy * (k - 1) + j) + i] - org +
                in[sx * (sy * (k + 1) + j) + i] - org) * factor;
}

void calRegularization(double* in, double factor, double* out,
                       int sx, int sy, int sz)
{
    int totalSize = sx * sy * sz;
    double* devT = 0;
    cudaMalloc(&devT, sizeof(double) * totalSize);
    cudaMemcpy(devT, in, sizeof(double) * totalSize,
    cudaMemcpyHostToDevice);

    // temps
    double* devReg = 0;
    cudaMalloc(&devReg, sizeof(double) * totalSize);
    cudaMemset(devReg, 0, sizeof(double) * totalSize);

    dim3 blockSize(4, 4, 4);
    // compute capability >= 2.x
    //dim3 gridSize((sx + blockSize.x - 1) / blockSize.x,
    //              (sy + blockSize.y - 1) / blockSize.y,
    //              (sz + blockSize.z - 1) / blockSize.z);
```

```
    // compute capabiltiy < 2.x
    int gy = (sy + blockSize.y - 1) / blockSize.y;
    int gz = (sz + blockSize.z - 1) / blockSize.z;
    dim3 gridSize((sx + blockSize.x - 1) / blockSize.x, gy * gz, 1);

    calRegularizationKernel<<<gridSize, blockSize>>>(devT, factor, devReg,
    sx, sy, sz);

    cudaMemcpy(out, devReg, sizeof(double) * totalSize, cudaMemcpyDeviceToHost);
    cudaFree(devReg);
    cudaFree(devT);
}
```

可以通过在 MATLAB 命令窗口使用 mex 命令创建 c-mex 文件。但是，首先要使用 nvcc 编译器编译 CUDA 函数，并在调用 mex 时链接到目标。你可以创建 c-mex 文件，如下所示：

- 在 Mac OS X 操作系统中

```
% mac
system('/Developer/NVIDIA/CUDA-5.0/bin/nvcc -c calRegularization.cu
-m64 -Xptxas -v');
mex cal_regularization_cuda.cpp calRegularization.o -lcudart -L"/
Developer/NVIDIA/CUDA-5.0/lib" -I"/Developer/NVIDIA/CUDA-5.0/
include" -v
```

- 在 Windows 64 位操作系统中

```
% MS Windows
system('nvcc -c calRegularization.cu -Xptxas -v');
mex cal_regularization_cuda.cpp calRegularization.obj -lcudart -L"C:
\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\lib\x64" -I"C:
\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\include" -v
```

如果创建 Microsoft C 编译器路径有问题，可以明确地设置编译器路径如下：

```
% MS Windows
system('nvcc -c calRegularization.cu -Xptxas -v -ccbin "C:\Program Files
(x86)\Microsoft Visual Studio 10.0\VC\bin"');
```

然后，稍微修改 deformableRegister3D(...)函数来调用 c-mex 版本。修改后函数为 deformableRegister3D_partialCuda.m。

```
function [movVol_updated, Tx, Ty, Tz] = deformableRegister3D_partial
Cuda(movVol, refVol, iterLimit)
[refX, refY, refZ] = size(refVol)
[Ty,Tx,Tz] = meshgrid(1:refY,1:refX,1:refZ);

% calculate gradient for moving volume
```

```
[movX, movY, movZ] = size(movVol)
movXGrad_Org = zeros(movX, movY, movZ);
movYGrad_Org = zeros(movX, movY, movZ);
movZGrad_Org = zeros(movX, movY, movZ);

movXGrad = zeros(movX, movY, movZ);
movYGrad = zeros(movX, movY, movZ);
movZGrad = zeros(movX, movY, movZ);

movXGrad_Org(2:end-1,2:end-1,2:end-1) = movVol(3:end,2:end-1,2:end-1) - movVol(1:end-2,2:end-1,2:end-1);
movYGrad_Org(2:end-1,2:end-1,2:end-1) = movVol(2:end-1,3:end,2:end-1) - movVol(2:end-1,1:end-2,2:end-1);
movZGrad_Org(2:end-1,2:end-1,2:end-1) = movVol(2:end-1,2:end-1,3:end) - movVol(2:end-1,2:end-1,1:end-2);

forceX = zeros(refX, refY, refZ);
forceY = zeros(refX, refY, refZ);
forceZ = zeros(refX, refY, refZ);

regX = zeros(refX, refY, refZ);
regY = zeros(refX, refY, refZ);
regZ = zeros(refX, refY, refZ);

delta_Tx = zeros(refX, refY, refZ);
delta_Ty = zeros(refX, refY, refZ);
delta_Tz = zeros(refX, refY, refZ);

for iter = 1:iterLimit
iter
movVol_updated = interp3(movVol, Ty, Tx, Tz); % make movVol in the ref space

movXGrad = interp3(movXGrad_Org, Ty, Tx, Tz);
movYGrad = interp3(movYGrad_Org, Ty, Tx, Tz);
movZGrad = interp3(movZGrad_Org, Ty, Tx, Tz);
% calculate deformation force
forceX = cal_deformation_force(refVol, movVol_updated, movXGrad);
forceY = cal_deformation_force(refVol, movVol_updated, movYGrad);
forceZ = cal_deformation_force(refVol, movVol_updated, movZGrad);
% Regularization
% Regularization makes the registration smoother warps.
regulFactor = 0.1;
regX = cal_regularization_cuda(Tx, regulFactor);
regY = cal_regularization_cuda(Ty, regulFactor);
```

```
regZ = cal_regularization_cuda(Tz, regulFactor);

% stepSize = 0.5;
stepSize = 1.0;
delta_Tx = cal_delta(forceX, regX, stepSize);
delta_Ty = cal_delta(forceY, regY, stepSize);
delta_Tz = cal_delta(forceZ, regZ, stepSize);

Tx = Tx + delta_Tx;
Ty = Ty + delta_Ty;
Tz = Tz + delta_Tz;

Tx = max(min(Tx, refX), 1);
Ty = max(min(Ty, refY), 1);
Tz = max(min(Tz, refZ), 1);
end
```

将 cal_regularization 转换为 CUDA 版本后，右海马体和左海马体分割的总运行时间都减少为 182s（见图 8.18），比 atlasSeg_Main.m 快大约 3.34 倍。如前所述，对 cal_regularization 单独进行 CUDA 转换中，循环内 CPU 和 GPU 存储之间的数据传输太频繁。8.5 节中，我们将进行整个运算的 CUDA 转换。

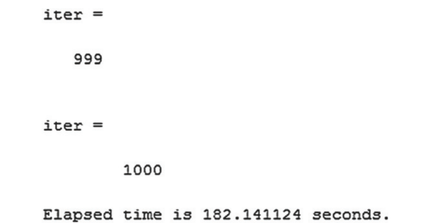


图 8.18　atlasSeg_PartialCuda.m 运行时间

8.5　CUDA 转换 2——图像配准

本节中，我们试图将 deformableRegister3D 中的全部迭代循环替换为 c-mex 和 CUDA 函数。为了最小化主机和设备之间的数据传输，将循环中的每个操作都改为 CUDA 调用。共创建 5 个 CUDA 内核函数。

MATLAB	CUDA 内核函数
interp3(...)	calInterp3<<<...>>>
cal_deformation_force	calDeformationForce<<<...>>>
cal_delta	calDelta<<<...>>>
cal_regularization	calRegularization<<<...>>>
Tx = Tx + delta_Tx; ... Tx = max(min(Tx, refX), 1);	updatePos<<<...>>>

我们的线程块由 4×4×4 线程组成，即每个线程块共 64 线程。线程网格大小是基于线程块尺寸和列尺寸。确定线程网格和线程块的大小为：

```
dim3 blockSize(4, 4, 4);

// compute capability >= 2.x
//dim3 gridSize((sx + blockSize.x - 1) / blockSize.x,
//  (sy + blockSize.y - 1) / blockSize.y,
//  (sz + blockSize.z - 1) / blockSize.z);

// compute capabiltiy < 2.x
int gy = (sy + blockSize.y - 1) / blockSize.y;
int gz = (sz + blockSize.z - 1) / blockSize.z;
dim3 gridSize((sx + blockSize.x - 1) / blockSize.x,
              gy * gz,
              1);
```

注意，根据 GPU 的计算能力，线程网格大小可能支持，也可能不支持三维。当不支持三维线程网格时，通过二维进行传输，然后如第 7 章所做的，在内核函数中恢复它们。

```
int i = blockIdx.x * blockDim.x + threadIdx.x;

// compute capability >= 2.x
//int j = blockIdx.y * blockDim.y + threadIdx.y;
//int k = blockIdx.z * blockDim.z + threadIdx.z;

// compute capability < 2.x
int gy = (sy + blockDim.y - 1) / blockDim.y;
int j = (blockIdx.y % gy) * blockDim.y + threadIdx.y;
int k = (blockIdx.y / gy) * blockDim.z + threadIdx.z;
```

一旦根据体积确定线程网格和块的大小，在 GPU 设备上给输入和输出分配存储空间。然后使用 cudaMemcpy(...) 从主机向 GPU 设备复制数据。在迭代循环中调用内核函数。在特定数量的迭代后，将结果传回主机。

共有两个主要文件：register3D.cpp 和 register3D_cuda.cu。register3D.cpp 执行 c-mex 的子例行程序且调用 register3D(…) 函数。register3D_cuda.cu 执行所有 CUDA 内核函数和 GPU 的存储操作。

当定义 CUDA 内核函数后，基于 GPU 运算能力内核函数可能不接受双精度浮点运算。需要确保你的 GPU 运算能力支持双精度。否则，需要将数据转换为单精度。本例中假定支持双精度。

registger3D.cpp

```
#include "mex.h"
#include <cuda_runtime.h>
#include "register3D_cuda.h"

// funciton [movVol_updated, Tx, Ty, Tz] = register3D(movVol, refVol, ...
//                         movXGrad_Org, movYGrad_Org, movZGrad_Org, ...
//                           Tx, Ty, Tz, ...
//                           iterLimit, regulFactor, stepSize);
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    if (nrhs != 11)
        mexErrMsgTxt("Invaid number of input arguments");

    if (nlhs != 4)
        mexErrMsgTxt("Invalid number of outputs");

    for (int i = 0; i < nrhs; ++i)
    {
        if (!mxIsDouble(prhs[i]))
        {
            mexErrMsgTxt("input vector data type must be double");
            break;
        }
    }

    const mwSize* size = mxGetDimensions(prhs[0]);
    int sx = size[0];
    int sy = size[1];
    int sz = size[2];

    // inputs
    double* movVol = (double*)mxGetData(prhs[0]);
    double* refVol = (double*)mxGetData(prhs[1]);
    double* movXGrad_Org = (double*)mxGetData(prhs[2]);
    double* movYGrad_Org = (double*)mxGetData(prhs[3]);
    double* movZGrad_Org = (double*)mxGetData(prhs[4]);
    double* TxIn = (double*)mxGetData(prhs[5]);
    double* TyIn = (double*)mxGetData(prhs[6]);
    double* TzIn = (double*)mxGetData(prhs[7]);
    double iterLimit = *((double*)mxGetData(prhs[8]));
    double regulFactor = *((double*)mxGetData(prhs[9]));
    double stepSize = *((double*)mxGetData(prhs[10]));
```

```

    // outputs
    plhs[0] = mxCreateNumericArray(3, size, mxDOUBLE_CLASS, mxREAL);
    double* movVol_updated = (double*)mxGetData(plhs[0]);
    plhs[1] = mxCreateNumericArray(3, size, mxDOUBLE_CLASS, mxREAL);
    double* Tx = (double*)mxGetData(plhs[1]);
    plhs[2] = mxCreateNumericArray(3, size, mxDOUBLE_CLASS, mxREAL);
    double* Ty = (double*)mxGetData(plhs[2]);
    plhs[3] = mxCreateNumericArray(3, size, mxDOUBLE_CLASS, mxREAL);
    double* Tz = (double*)mxGetData(plhs[3]);

    // register3D cuda
    register3D_cuda(movVol,
                    refVol,
                    movXGrad_Org,
                    movYGrad_Org,
                    movZGrad_Org,
                    TxIn, TyIn, TzIn,
                    movVol_updated,
                    Tx, Ty, Tz,
                    sx, sy, sz,
                    (int)iterLimit, regulFactor, stepSize);

    cudaError_t error = cudaGetLastError();
    if (error != cudaSuccess)
    {
        mexPrintf("%s\n", cudaGetErrorString(error));
        mexErrMsgTxt("CUDA failed\n");
    }
}
```

register3D_cuda.h

```
#ifndef __DEFORMABLEREGISTER3D_H__
#define __DEFORMABLEREGISTER3D_H__

extern void register3D_cuda(double* movVol, // in
                            double* refVol, // in
                            double* movXGrad_Org, // in
                            double* movYGrad_Org, // in
                            double* movZGrad_Org, // in
                            double* TxIn,  // in
                            double* TyIn,  // in
                            double* TzIn,  // in
                            double* movVol_updated, // out
                            double* Tx,  // out
                            double* Ty,  // out
```

```
                                double* Tz,   // out
                                int sx, int sy, int sz,
                                int iterLimit,
                                double regulFacotr,
                                double stepSize);

#endif // __DEFORMABLEREGISTER3D_H__
```

register3D_cuda.cu

```
#include "register3D_cuda.h"

__global__ void calInterp3(double* Vin,
                                double* Tx, double* Ty, double* Tz,
                                double* Vout,
                                int sx, int sy, int sz)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    // compute capability >= 2.x
    //int j = blockIdx.y * blockDim.y + threadIdx.y;
    //int k = blockIdx.z * blockDim.z + threadIdx.z;
    // compute capability < 2.x
    int gy = (sy + blockDim.y - 1) / blockDim.y;
    int j = (blockIdx.y % gy) * blockDim.y + threadIdx.y;
    int k = (blockIdx.y / gy) * blockDim.z + threadIdx.z;

    if (i >= sx - 1 || j >= sy - 1 || k >= sz - 1)
       return;

    int idx = sx * (sy * k + j) + i;

    double tx = Tx[idx];
    double ty = Ty[idx];
    double tz = Tz[idx];
    int ix = (int)tx - 1;
    int iy = (int)ty - 1;
    int iz = (int)tz - 1;
    int idx0 = sx * (sy * iz + iy) + ix;
    int idx1 = sx * (sy * iz + iy) + ix + 1;
    int idx2 = sx * (sy * iz + iy + 1) + ix;
    int idx3 = sx * (sy * iz + iy + 1) + ix + 1;
    int idx4 = sx * (sy * (iz + 1) + iy) + ix;
    int idx5 = sx * (sy * (iz + 1) + iy) + ix + 1;
    int idx6 = sx * (sy * (iz + 1) + iy + 1) + ix;
    int idx7 = sx * (sy * (iz + 1) + iy + 1) + ix + 1;

    // along x
```

```
    double wd = floor(tx + 1.0) - tx;
    double wu = 1.0 - wd;
    double R00 = wd * Vin[idx0] + wu * Vin[idx1];
    double R10 = wd * Vin[idx2] + wu * Vin[idx3];
    double R01 = wd * Vin[idx4] + wu * Vin[idx5];
    double R11 = wd * Vin[idx6] + wu * Vin[idx7];

    // along y
    wd = floor(ty + 1.0) - ty;
    wu = 1.0 - wd;
    double R0 = wd * R00 + wu * R10;
    double R1 = wd * R01 + wu * R11;

    // along z
    wd = floor(tz + 1.0) - tz;
    wu = 1.0 - wd;

    Vout[idx] = wd * R0 + wu * R1;
}

__global__ void calDeformationForce(double* refImg, double* movImg,
                                    double* gradientImg, double* forceImg,
                                    int sx, int sy, int sz)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    // compute capability >= 2.x
    //int j = blockIdx.y * blockDim.y + threadIdx.y;
    //int k = blockIdx.z * blockDim.z + threadIdx.z;
    // compute capability < 2.x
    int gy = (sy + blockDim.y - 1) / blockDim.y;
    int j = (blockIdx.y % gy) * blockDim.y + threadIdx.y;
    int k = (blockIdx.y / gy) * blockDim.z + threadIdx.z;

    if (i >= sx || j >= sy || k >= sz)
        return;

    int idx = sx * (sy * k + j) + i;
    forceImg[idx] = (refImg[idx] - movImg[idx]) * gradientImg[idx];
}

__global__ void calDelta(double* deformForce, double* resistance,
                         double* delta, double stepSize,
                         int sx, int sy, int sz)
{
     int i = blockIdx.x * blockDim.x + threadIdx.x;
     // compute capability >= 2.x
```

```
    //int j = blockIdx.y * blockDim.y + threadIdx.y;
    //int k = blockIdx.z * blockDim.z + threadIdx.z;
    // compute capability < 2.x
    int gy = (sy + blockDim.y - 1) / blockDim.y;
    int j = (blockIdx.y % gy) * blockDim.y + threadIdx.y;
    int k = (blockIdx.y / gy) * blockDim.z + threadIdx.z;

    if (i >= sx || j >= sy || k >= sz)
        return;

    int idx = sx * (sy * k + j) + i;

    double temp = (deformForce[idx] + resistance[idx]) * stepSize;
    temp = (temp > -1.0) ? temp : -1.0; // max
    delta[idx] = (temp < 1.0) ? temp : 1.0; // min
}

__global__ void updatePos (double* deltaX, double* deltaY, double*
                                deltaZ,
                                double* Tx, double* Ty, double* Tz,
                                int sx, int sy, int sz)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    // compute capability >= 2.x
    //int j = blockIdx.y * blockDim.y + threadIdx.y;
    //int k = blockIdx.z * blockDim.z + threadIdx.z;
    // compute capability < 2.x
    int gy = (sy + blockDim.y - 1) / blockDim.y;
    int j = (blockIdx.y % gy) * blockDim.y + threadIdx.y;
    int k = (blockIdx.y / gy) * blockDim.z + threadIdx.z;

    if (i >= sx || j >= sy || k >= sz)
        return;

    int idx = sx * (sy * k + j) + i;

    double tempX = Tx[idx] + deltaX[idx];
    double tempY = Ty[idx] + deltaY[idx];
    double tempZ = Tz[idx] + deltaZ[idx];

    tempX = (tempX < sx) ? tempX : sx; // min
    tempY = (tempY < sy) ? tempY : sy; // min
    tempZ = (tempZ < sz) ? tempZ : sz; // min

    Tx[idx] = (tempX > 1.0) ? tempX : 1.0; // max
    Ty[idx] = (tempY > 1.0) ? tempY : 1.0; // max
    Tz[idx] = (tempZ > 1.0) ? tempZ : 1.0; // max
```

```
}

__global__ void calRegularization(double* in, double factor, double* out,
                                  int sx, int sy, int sz)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    // compute capability >= 2.x
    //int j = blockIdx.y * blockDim.y + threadIdx.y;
    //int k = blockIdx.z * blockDim.z + threadIdx.z;
    // compute capability < 2.x
    int gy = (sy + blockDim.y - 1) / blockDim.y;
    int j = (blockIdx.y % gy) * blockDim.y + threadIdx.y;
    int k = (blockIdx.y / gy) * blockDim.z + threadIdx.z;

    if (i < 1 || i >= sx - 1 ||
        j < 1 || j >= sy - 1 ||
        k < 1 || k >= sz - 1)
        return;

    int idx = sx * (sy * k + j) + i;
    double org = in[idx];
    out[idx] = (in[sx * (sy * k + j) + i - 1] - org +
                in[sx * (sy * k + j) + i + 1] - org +
                in[sx * (sy * k + j - 1) + i] - org +
                in[sx * (sy * k + j + 1) + i] - org +
                in[sx * (sy * (k - 1) + j) + i] - org +
                in[sx * (sy * (k + 1) + j) + i] - org) * factor;
}


void register3D_cuda(double* movVol, // in
                     double* refVol, // in
                     double* movXGrad_Org, // in
                     double* movYGrad_Org, // in
                     double* movZGrad_Org, // in
                     double* TxIn,  // in
                     double* TyIn,  // in
                     double* TzIn,  // in
                     double* movVol_updated, // out
                     double* Tx,  // out
                     double* Ty,  // out
                     double* Tz,  // out
                     int sx, int sy, int sz,
                     int iterLimit,
                     double regulFactor,
                     double stepSize)
```

```
{
    int totalSize = sx * sy * sz;

    // inputs
    double* devMovVol = 0;
    double* devRefVol = 0;
    double* devMovXGrad_Org = 0;
    double* devMovYGrad_Org = 0;
    double* devMovZGrad_Org = 0;
    cudaMalloc(&devMovVol, sizeof(double) * totalSize);
    cudaMalloc(&devRefVol, sizeof(double) * totalSize);
    cudaMalloc(&devMovXGrad_Org, sizeof(double) * totalSize);
    cudaMalloc(&devMovYGrad_Org, sizeof(double) * totalSize);
    cudaMalloc(&devMovZGrad_Org, sizeof(double) * totalSize);
    cudaMemcpy(devMovVol, movVol, sizeof(double) * totalSize,
               cudaMemcpyHostToDevice);
    cudaMemcpy(devRefVol, refVol, sizeof(double) * totalSize,
               cudaMemcpyHostToDevice);
    cudaMemcpy(devMovXGrad_Org, movXGrad_Org, sizeof(double) *
               totalSize, cudaMemcpyHostToDevice);
    cudaMemcpy(devMovYGrad_Org, movYGrad_Org, sizeof(double) *
               totalSize, cudaMemcpyHostToDevice);
    cudaMemcpy(devMovZGrad_Org, movZGrad_Org, sizeof(double) *
               totalSize, cudaMemcpyHostToDevice);

    // temps
    double* devMovXGrad = 0;
    double* devMovYGrad = 0;
    double* devMovZGrad = 0;
    double* devForceX = 0;
    double* devForceY = 0;
    double* devForceZ = 0;
    double* devRegX = 0;
    double* devRegY = 0;
    double* devRegZ = 0;
    double* devDeltaTx = 0;
    double* devDeltaTy = 0;
    double* devDeltaTz = 0;
    cudaMalloc(&devMovXGrad, sizeof(double) * totalSize);
    cudaMalloc(&devMovYGrad, sizeof(double) * totalSize);
    cudaMalloc(&devMovZGrad, sizeof(double) * totalSize);
    cudaMalloc(&devForceX, sizeof(double) * totalSize);
    cudaMalloc(&devForceY, sizeof(double) * totalSize);
    cudaMalloc(&devForceZ, sizeof(double) * totalSize);
    cudaMalloc(&devRegX, sizeof(double) * totalSize);
    cudaMalloc(&devRegY, sizeof(double) * totalSize);
    cudaMalloc(&devRegZ, sizeof(double) * totalSize);
    cudaMalloc(&devDeltaTx, sizeof(double) * totalSize);
```

```
cudaMalloc(&devDeltaTy, sizeof(double) * totalSize);
cudaMalloc(&devDeltaTz, sizeof(double) * totalSize);
cudaMemset(devMovXGrad, 0, sizeof(double) * totalSize);
cudaMemset(devMovYGrad, 0, sizeof(double) * totalSize);
cudaMemset(devMovZGrad, 0, sizeof(double) * totalSize);
cudaMemset(devForceX, 0, sizeof(double) * totalSize);
cudaMemset(devForceY, 0, sizeof(double) * totalSize);
cudaMemset(devForceZ, 0, sizeof(double) * totalSize);
cudaMemset(devRegX, 0, sizeof(double) * totalSize);
cudaMemset(devRegY, 0, sizeof(double) * totalSize);
cudaMemset(devRegZ, 0, sizeof(double) * totalSize);
cudaMemset(devDeltaTx, 0, sizeof(double) * totalSize);
cudaMemset(devDeltaTy, 0, sizeof(double) * totalSize);
cudaMemset(devDeltaTz, 0, sizeof(double) * totalSize);

// outputs
double* devMovVol_updated = 0;
double* devTx = 0;
double* devTy = 0;
double* devTz = 0;
cudaMalloc(&devMovVol_updated, sizeof(double) * totalSize);
cudaMalloc(&devTx, sizeof(double) * totalSize);
cudaMalloc(&devTy, sizeof(double) * totalSize);
cudaMalloc(&devTz, sizeof(double) * totalSize);
cudaMemcpy(devMovVol_updated, movVol, sizeof(double) *
totalSize, cudaMemcpyHostToDevice);

// init Tx, Ty, Tz
cudaMemcpy(devTx, TxIn, sizeof(double) * totalSize,
           cudaMemcpyHostToDevice);
cudaMemcpy(devTy, TyIn, sizeof(double) * totalSize,
           cudaMemcpyHostToDevice);
cudaMemcpy(devTz, TzIn, sizeof(double) * totalSize,
           cudaMemcpyHostToDevice);

dim3 blockSize(4, 4, 4);
// compute capability >= 2.x
//dim3 gridSize((sx + blockSize.x - 1) / blockSize.x,
//  (sy + blockSize.y - 1) / blockSize.y,
//  (sz + blockSize.z - 1) / blockSize.z);
// compute capabiltiy < 2.x
int gy = (sy + blockSize.y - 1) / blockSize.y;
int gz = (sz + blockSize.z - 1) / blockSize.z;
dim3 gridSize((sx + blockSize.x - 1) / blockSize.x,
              gy * gz,
              1);

```

```
    calInterp3<<<gridSize, blockSize>>>(devMovVol, devTx,
    devTy, devTz,
    devMovVol_updated, sx, sy, sz);

for (int i = 0; i < iterLimit; ++i)
{
    // interpolation
    calInterp3<<<gridSize, blockSize>>>(devMovVol, devTx,
    devTy, devTz,
    devMovVol_updated, sx, sy, sz);
    calInterp3<<<gridSize, blockSize>>>(devMovXGrad_Org,
    devTx, devTy, devTz,
    devMovXGrad, sx, sy, sz);
    calInterp3<<<gridSize, blockSize>>>(devMovYGrad_Org,
    devTx, devTy, devTz,
    devMovYGrad, sx, sy, sz);
    calInterp3<<<gridSize, blockSize>>>(devMovZGrad_Org,
    devTx, devTy, devTz,
    devMovZGrad, sx, sy, sz);

    // deformation force
    calDeformationForce<<<gridSize, blockSize>>>(devRefVol,
    devMovVol_updated,
    devMovXGrad, devForceX,
    sx, sy, sz);
    calDeformationForce<<<gridSize, blockSize>>>(devRefVol,
    devMovVol_updated,
    devMovYGrad, devForceY,
    sx, sy, sz);
    calDeformationForce<<<gridSize, blockSize>>>(devRefVol,
    devMovVol_updated,
    devMovZGrad, devForceZ,
    sx, sy, sz);

    // Regularization
    calRegularization<<<gridSize, blockSize>>>(devTx,
    regulFactor, devRegX, sx, sy, sz);
    calRegularization<<<gridSize, blockSize>>>(devTy,
    regulFactor, devRegY, sx, sy, sz);
    calRegularization<<<gridSize, blockSize>>>(devTz,
    regulFactor, devRegZ, sx, sy, sz);

    // Calculate delta
    double stepSize = 1.0;
    calDelta<<<gridSize, blockSize>>>(devForceX, devRegX,
    devDeltaTx, stepSize, sx, sy, sz);
    calDelta<<<gridSize, blockSize>>>(devForceY, devRegY,
    devDeltaTy, stepSize, sx, sy, sz);
```

```
    calDelta<<<gridSize, blockSize>>>(devForceZ, devRegZ, devDeltaTz, stepSize, sx, sy, sz);

    // Update pos
    updatePos<<<gridSize, blockSize>>>(devDeltaTx, devDeltaTy, devDeltaTz,
    devTx, devTy, devTz, sx, sy, sz);

    //error = cudaGetLastError();
    //if (error != cudaSuccess)
    // exit(-1);
}


    // copy outputs
    cudaMemcpy(movVol_updated, devMovVol_updated, sizeof(double) * totalSize, cudaMemcpyDeviceToHost);
    cudaMemcpy(Tx, devTx, sizeof(double) * totalSize, cudaMemcpyDeviceToHost);
    cudaMemcpy(Ty, devTy, sizeof(double) * totalSize, cudaMemcpyDeviceToHost);
    cudaMemcpy(Tz, devTz, sizeof(double) * totalSize, cudaMemcpyDeviceToHost);

    // free resources
    cudaFree(devMovVol);
    cudaFree(devRefVol);
    cudaFree(devMovXGrad_Org);
    cudaFree(devMovYGrad_Org);
    cudaFree(devMovZGrad_Org);

    cudaFree(devMovXGrad);
    cudaFree(devMovYGrad);
    cudaFree(devMovZGrad);
    cudaFree(devForceX);
    cudaFree(devForceY);
    cudaFree(devForceZ);
    cudaFree(devRegX);
    cudaFree(devRegY);
    cudaFree(devRegZ);
    cudaFree(devDeltaTx);
    cudaFree(devDeltaTy);
    cudaFree(devDeltaTz);

    cudaFree(devMovVol_updated);
    cudaFree(devTx);
    cudaFree(devTy);
    cudaFree(devTz);
```

```
360 }
```

可以通过在 MATLAB 命令窗口使用 mex 命令创建 c-mex 文件。但首先要使用 nvcc 编译器编译 CUDA 函数并在调用 mex 时链接到目标。你可以创建 c-mex 文件如下所示：

- 在 Mac OS X 操作系统中

```
% mac
system('/Developer/NVIDIA/CUDA-5.0/bin/nvcc  -c  register3D_cuda.cu
-m64 -Xptxas -v');
mex register3D.cpp register3D_cuda.o -lcudart -L"/Developer/NVIDIA/
CUDA-5.0/lib" -I"/Developer/NVIDIA/CUDA-5.0/include" -v
```

- 在 Windows 64 位操作系统中

```
% MS Windows
system('nvcc -c register3D_cuda.cu -Xptxas -v');
mex register3D.cpp register3D_cuda.obj -lcudart -L"C:\Program Files
\NVIDIA GPU Computing Toolkit\CUDA\v5.0\lib\x64" -I"C:\Program Files
\NVIDIA GPU Computing Toolkit\CUDA\v5.0\include" -v
```

然后适当修改 deformableRegister3D(...) 函数，来调用 c-mex 版本。修改后的函数为 deformableRegister3D_cuda.m。

```
function  [movVol_updated,  Tx,  Ty,  Tz]  =  deformableRegister3D_cuda
(movVol, refVol, iterLimit)
[refX, refY, refZ] = size(refVol)
[Ty,Tx,Tz] = meshgrid(1:refY,1:refX,1:refZ);

% calculate gradient for moving volume
[movX, movY, movZ] = size(movVol)
movXGrad_Org = zeros(movX, movY, movZ);
movYGrad_Org = zeros(movX, movY, movZ);
movZGrad_Org = zeros(movX, movY, movZ);

movXGrad_Org(2:end-1,2:end-1,2:end-1)=movVol(3:end,2:end-1,2:
end-1)-movVol(1:end-2,2:end-1,2:end-1);
movYGrad_Org(2:end-1,2:end-1,2:end-1)=movVol(2:end-1,3:end,2:
end-1)-movVol(2:end-1,1:end-2,2:end-1);
movZGrad_Org(2:end-1,2:end-1,2:end-1)=movVol(2:end-1,2:end-1,3:
end)-movVol(2:end-1,2:end-1,1:end-2);
regulFactor = 0.1;
stepSize = 1.0;
[movVol_updated, Tx, Ty, Tz] = register3D(movVol, refVol, ...
                                movXGrad_Org, movYGrad_Org, movZGrad_Org, ...
                                Tx, Ty, Tz, ...
                                iterLimit, regulFactor, stepSize);
```

8.6 CUDA 转换结果

先从时间方面检查使用 GPU 获得了多少速度提升。在 Profiler 中运行 atlasSeg_Cuda.m，如图 8.19 所示。

```
movX =

    65


movY =

    52


movZ =

    55

Elapsed time is 26.080896 seconds.
```

图 8.19 atlasSeg_Cuda.m 运行时间

经 CUDA 转换后，atlasSeg_Cuda.m 的左海马体和右海马体处理时间大约为 26s，比 atlasSeg_Main.m 快约 24 倍。让我们看一下 CUDA 转换后分析结果发生了什么变化（见图 8.20）。

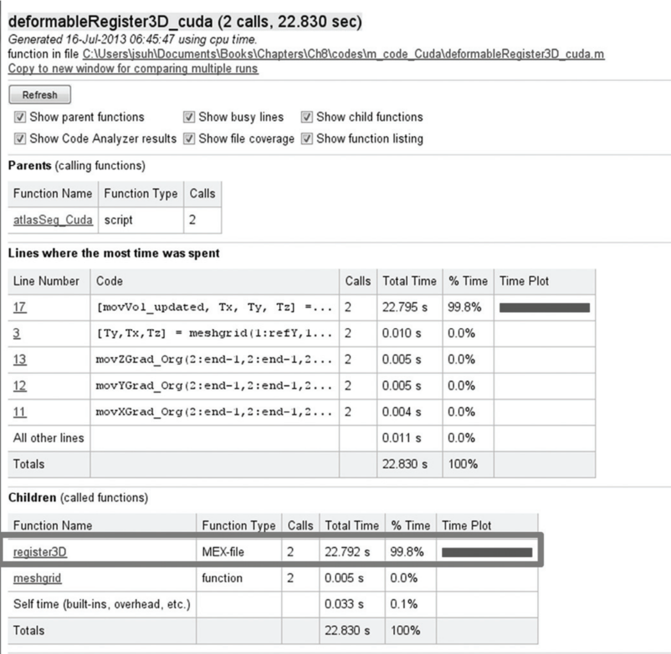
Profile Summary

Generated 16-Jul-2013 06:41:37 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
atlasSeg_Cuda	1	26.892 s	0.559 s	
deformableRegister3D_cuda	2	22.830 s	0.033 s	
register3D (MEX-file)	2	22.792 s	22.792 s	
crop_around_ROI	2	1.126 s	1.126 s	
imshow	84	0.872 s	0.119 s	
myView2	4	0.818 s	0.439 s	
myView_overlap_red	3	0.751 s	0.310 s	
myView_overlap_green	3	0.733 s	0.294 s	
newplot	168	0.414 s	0.036 s	
newplot>ObserveAxesNextPlot	168	0.370 s	0.017 s	
cla	132	0.353 s	0.008 s	
graphics\private\clo	132	0.345 s	0.155 s	
imuitools\private\basicImageDisplay	84	0.317 s	0.084 s	
subplot	66	0.190 s	0.067 s	

图 8.20 CUDA 转换后主函数（atlasSeg_Cuda.m）分析结果

与 atlasSeg_main.m 的分析结果相比（见图 8.12），deformableRegister3D_cuda 函数消耗了更少的时间。图 8.21 表明，在 deformableRegister3D_cuda 函数中 register3D 函数依旧占总处理时间的 99.8%，但总时间为 22.792s。图 8.22 为 register3D 模块的分析结果。由于 register3D 模块是 c-mex 模块，所以 MATLAB 分析器无法显示代码分析更多的信息。



deformableRegister3D_cuda (2 calls, 22.830 sec)
Generated 16-Jul-2013 06:45:47 using cpu time.
function in file C:\Users\jsuh\Documents\Books\Chapters\Ch8\codes\m_code_Cuda\deformableRegister3D_cuda.m
Copy to new window for comparing multiple runs

Refresh

Show parent functions　Show busy lines　Show child functions
Show Code Analyzer results　Show file coverage　Show function listing

Parents (calling functions)

Function Name	Function Type	Calls
atlasSeg_Cuda	script	2

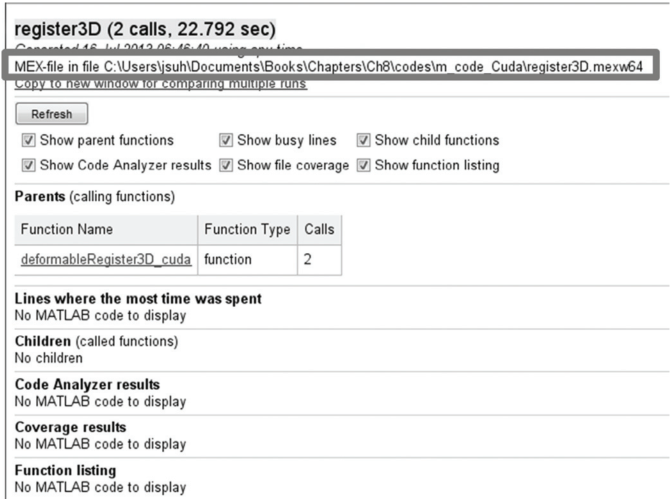
Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
17	[movVol_updated, Tx, Ty, Tz] =...	2	22.795 s	99.8%	
3	[Ty,Tx,Tz] = meshgrid(1:refY,1...	2	0.010 s	0.0%	
13	movZGrad_Org(2:end-1,2:end-1,2...	2	0.005 s	0.0%	
12	movYGrad_Org(2:end-1,2:end-1,2...	2	0.005 s	0.0%	
11	movXGrad_Org(2:end-1,2:end-1,2...	2	0.004 s	0.0%	
All other lines			0.011 s	0.0%	
Totals			22.830 s	100%	

Children (called functions)

Function Name	Function Type	Calls	Total Time	% Time	Time Plot
register3D	MEX-file	2	22.792 s	99.8%	
meshgrid	function	2	0.005 s	0.0%	
Self time (built-ins, overhead, etc.)			0.033 s	0.1%	
Totals			22.830 s	100%	

图 8.21　deformableRegister3D_cuda 模块分析结果



register3D (2 calls, 22.792 sec)
Generated 16-Jul-2013 06:46:40 using cpu time.
MEX-file in file C:\Users\jsuh\Documents\Books\Chapters\Ch8\codes\m_code_Cuda\register3D.mexw64
Copy to new window for comparing multiple runs

Refresh

Show parent functions　Show busy lines　Show child functions
Show Code Analyzer results　Show file coverage　Show function listing

Parents (calling functions)

Function Name	Function Type	Calls
deformableRegister3D_cuda	function	2

Lines where the most time was spent
No MATLAB code to display

Children (called functions)
No children

Code Analyzer results
No MATLAB code to display

Coverage results
No MATLAB code to display

Function listing
No MATLAB code to display

图 8.22　c-mex register3D 模块分析结果

8.7 结论

当对分析中显示为最忙的单个函数进行 CUDA 转换后，相比于 3D 医学图像分割的纯 m 代码速度提升大约 3.3 倍。然而，在每次迭代中，转换为 CUDA 的函数执行前后，CPU 和 GPU 存储之间进行过于频繁的数据传输。这使得整体进程效率下降。最终，对循环内全部运算都进行 CUDA 转换中，算法运行时间比 3D 医学图像分割的纯 m 代码快大约 24 倍。

速度优化的第一步是分析，从而明确要转换为 CUDA 函数的目标模块。第二步，当计划将 MATLAB 的 m 代码转换为 GPU 可用的 c-mex 代码时，应该仔细考虑数据的大小和 CPU 内存与 GPU 设备存储之间数据传输的频率。引入过多的、不必要的大数据传输可能无法达成速度提升的目的，这是因为主机与 GPU 设备之间的数据传输在 GPU 运算中是最慢的。

附　　录

附录 A　下载和安装 CUDA 库

A.1　CUDA 工具箱下载

要开始使用系统中的 CUDA，首先需要安装 CUDA 开发工具，并确保这些工具的正确操作。可以从 NVIDIA 网站http://www.nvidia.com/content/cuda/cuda-downloads.html处下载 CUDA 工具箱，如图 A.1 所示。

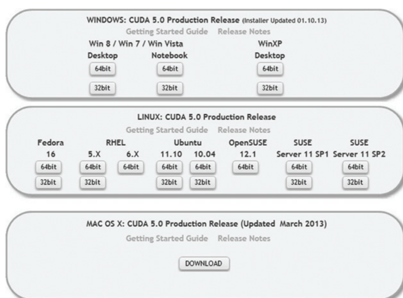


图 A.1　下载适用于不同操作系统的 CUDA

A.2　安装

根据所使用的操作系统，从图 A.1 中的链接选择下载安装程序。下载一旦完成，就可以执行安装程序，并按照屏幕上的提示开始安装，例如：

- 对于 64 位 Windows 7，如图 A.2 所示。

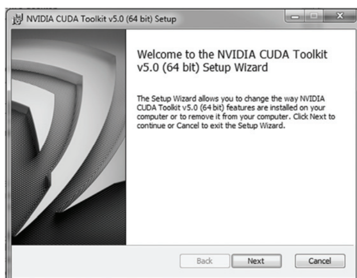


图 A.2　Windows CUDA 安装程序

- 对于 Mac OS X，如图 A.3 所示。

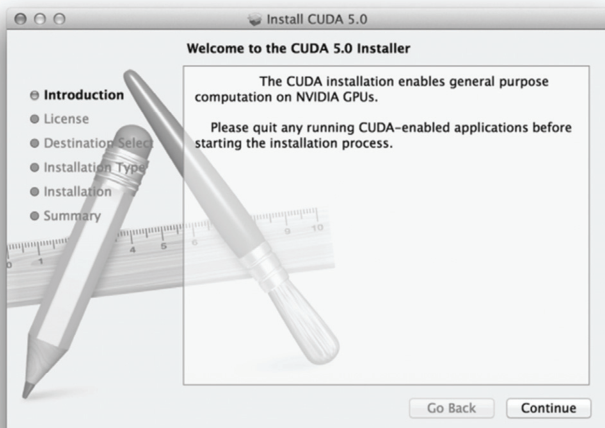


图 A.3　Mac OS X CUDA 安装程序

- 对于 Linux，如图 A.4 所示。

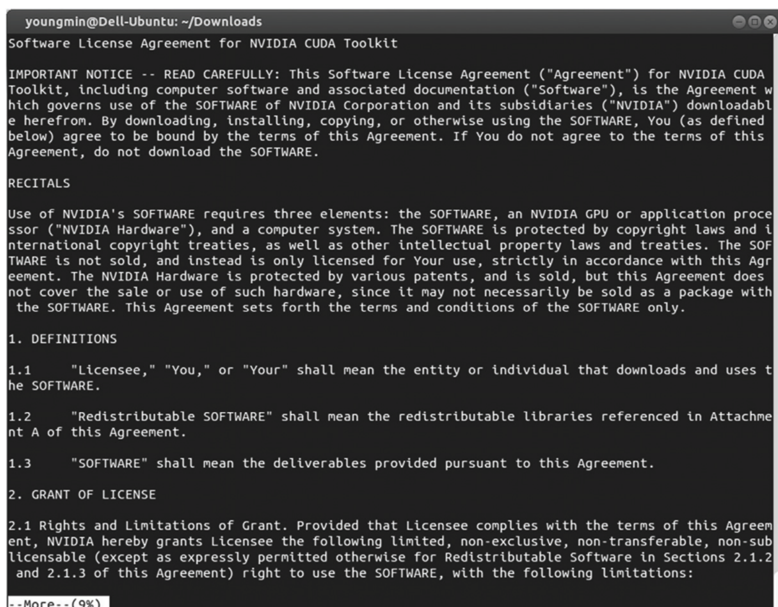


图 A.4　Linux CUDA 安装程序

安装完成后，要对 CUDA 是否正确安装与配置进行验证。了解 CUDA 工具箱的安装位置以及如何在系统上配置，可以令 CUDA 编程的体验更加愉悦。如果遵循默认安装选项，那么依据如下所示为每种系统安装 CUDA：

● 对于 Windows 7（64 位），默认安装目录为 C:\Program Files\NVIDIA GPU Computing Toolkit\ CUDA\v5.0，如图 A.5 所示。

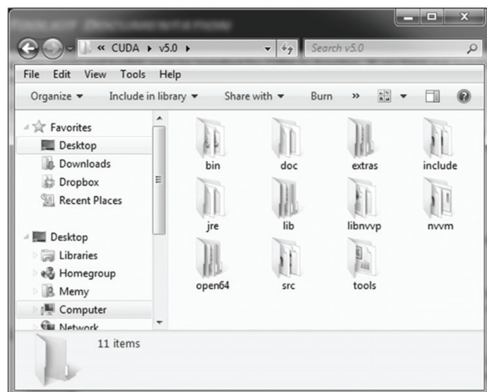
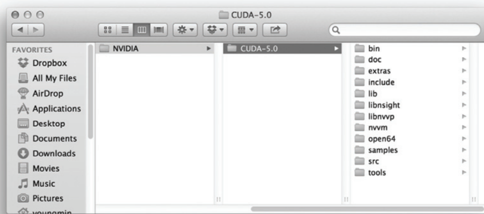


图 A.5　Windows CUDA 默认安装目录

● 对于 Mac OS X，默认安装目录为/Developer/NVIDIA/CUDA-5.0，如图 A.6 所示。

● 对于 Linux，默认安装目录为/usr/local/cuda-5.0，如图 A.7 所示。



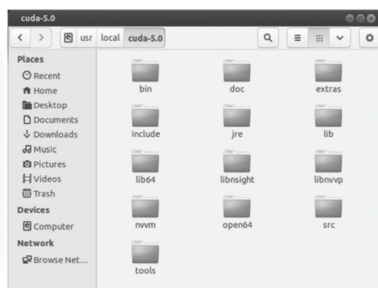


图 A.6　Mac OS X 默认安装目录　　图 A.7　Linux 默认安装目录（本例中为 Ubuntu）

对于每种操作系统，找出以下子目录下的文件。

● bin 下的 nvcc.exe 或 nvcc。
● include 下的 cuda_runtime.h。
● 对于 Windows 7（64 位），lib\64 下的 cudart.lib。
● 对于 Linux，lib 下的 libcudart.so。
● 对于 Mac OS X，lib 下的 libcudart.dylib。

A.3 确认

现在，最重要的是能够通过命令提示符执行 CUDA 编译器 nvcc。打开 Windows 中的命令提示符窗口，或者 Linux 和 Mac OS X 中的 shell。在提示符下执行图 A.8 中的指令。如果不能在命令提示符运行 nvcc，则确保安装的 CUDA 工具箱的 bin 目录处于系统环境中。

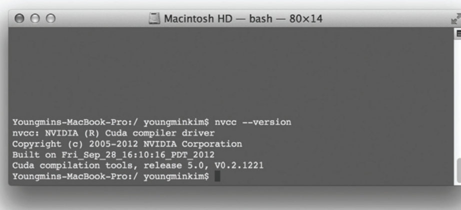


图 A.8 nvcc 版本验证

- 对于 Windows 7 64 位，在命令提示符下输入 PATH，然后看看带有 nvcc 的目录是否同图 A.9 中规定的一样。

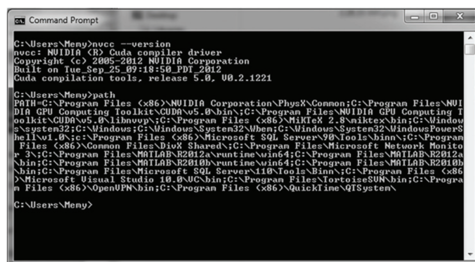


图 A.9 Windows 环境中的 nvcc PATH

- 对于 Mac OS X 与 Linux，在 shell 中的提示符下键入 echo $PATH，确保 nvcc 路径同图 A.10 与图 A.11 中规定的相同。

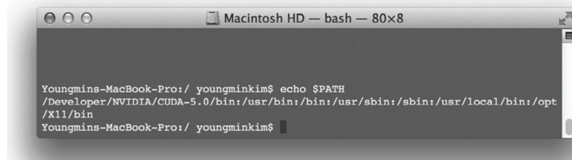


图 A.10 Mac OS X 中的 nvcc PATH

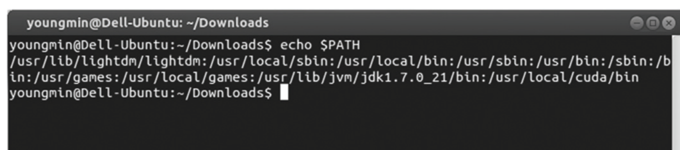


图 A.11 Linux 环境中的 nvcc PATH

附录 B　安装 NVIDIA Nsight 到 Visual Studio

NVIDIA Nsight（Visual Studio 版本）是 CUDA 的开发环境。NVIDIA Nsight（Visual Studio 版本）提供了强大的调试与分析功能，对 CUDA 代码开发极为有效。虽然 NVIDIA Nsight 是免费的，不过仍需要注册下载。

1. 访问 NVIDIA Nsight (http://www.nvidia.com/object/nsight.html) 进行用户注册，会看到图 B.1 中的屏幕。

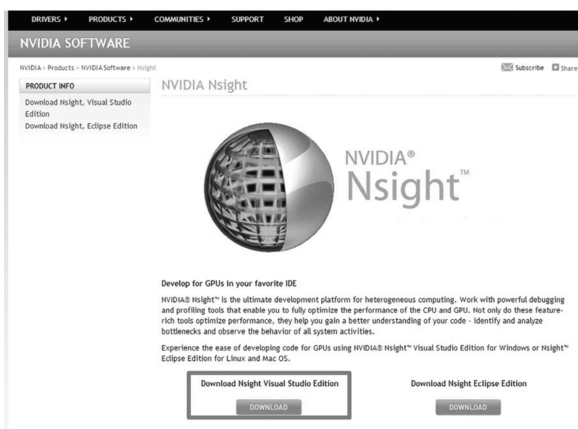


图 B.1　NVIDIA Nsight 网站。选择 Download Nsight Visual Studio Edition for Visual Studio

2. 注册后，下载与你的操作系统匹配的工具，如图 B.2 所示。

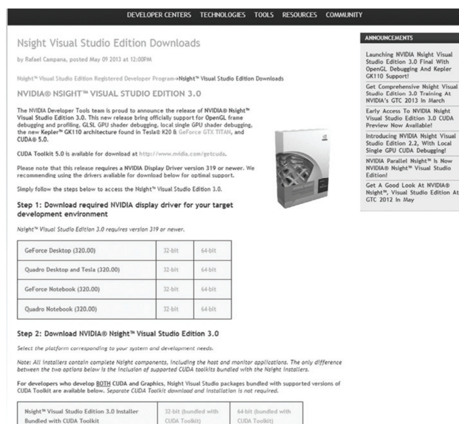


图 B.2　注册后的下载网站。可以根据操作系统下载选择版本

3. 如图 B.3 所示，安装 NVIDIA Nsight（Visual Studio 版本）。

4. 安装完成后（见图 B.4），在 Microsoft Visual Studio 中找到 Nsight 菜单。如

果看到这个菜单（见图 B.4），那么 Nsight 的全部安装进程就顺利完成了。

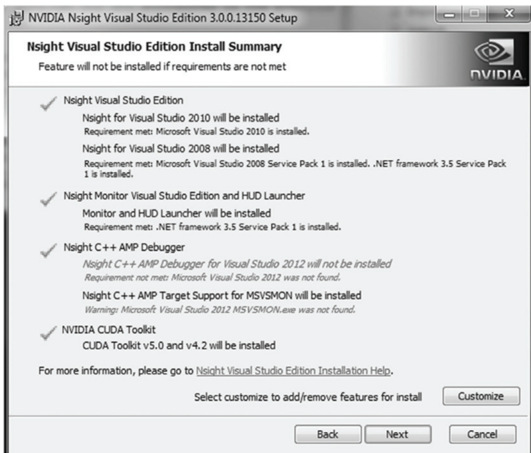


图 B.3 NVIDIA Nsight（Visual Studio 版本）的安装窗口

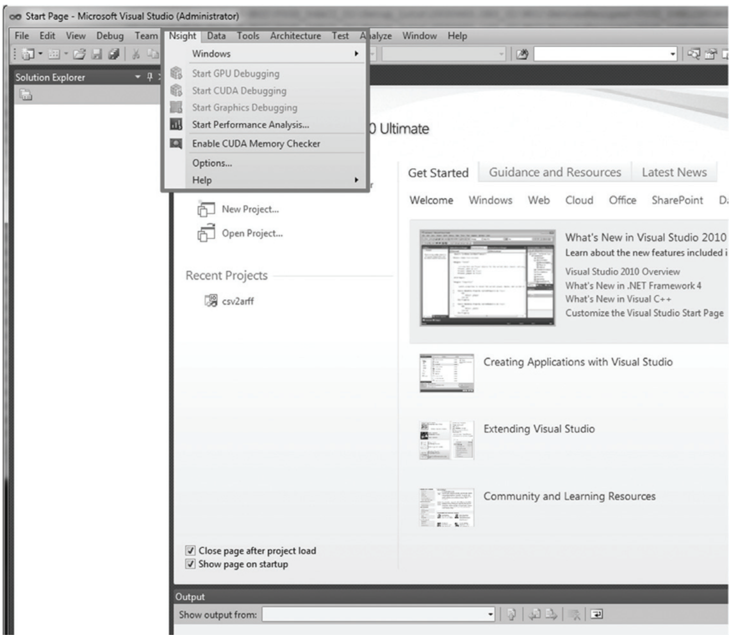


图 B.4 成功安装后，在 Microsoft Visual Studio 中查看 Nsight 菜单

国际视野　科技前沿

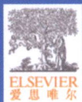
关于本书

除了仿真和算法开发，当前越来越多的研发人员使用MATLAB进行复杂计算领域的产品部署。用户可以借助图像处理器分布式并行处理，提升MATLAB代码的性能。由于提供了很多高层函数，MATLAB成功成为用于快速原型设计的出色仿真工具。但面对纷繁复杂的GPU细节和背景知识，MATLAB用户在面对GPU强大计算能力时，总是犹豫不决。本书为用户提供了入门读物，架起了MATLAB和GPU之间的桥梁。本书从零基础开始，深入浅出，如介绍MATLAB使用CUDA所需的设置（支持Windows、Linux和Mac OX等多种操作系统），引导用户通过一个个的专题（如CDUA库），逐步掌握GPU编程。作者还与读者分享了在大数据计算领域的MATLAB、C++和GPU的编程经验，展示了如何修改MATLAB代码以更好地利用GPU的计算能力，以及如何将代码整合到商用软件产品中。全书提供了大量的代码示例，能够作为用户C-MEX和CUDA代码的模板。

关于作者

Jung W. Suh 美国KLA-Tencor（科天）公司的高级算法工程师和研究科学家。2007年因其在3D医学图像处理领域的工作，从弗吉尼亚理工大学获得博士学位。他参与了三星电子在MPEG-4和数字移动广播（DMB）系统的研发工作。在任职KLA-Tencor公司前，他还担任HeartFlow公司高级科学家。研究领域包括生物图像处理、模式识别、机器学习和图像/视频压缩。发表30余篇期刊和会议论文，并拥有6项专利。

Youngmin Kim 美国Life Technologies（生命科技）公司的资深软件工程师，从事实时图像获取和高吞吐量图像分析程序开发工作。他之前的工作还包括设计和开发自动显微镜和用于实时分析的集成成像算法软件。先后从伊利诺伊大学（厄巴纳－香槟校区）电子工程专业获得学士和硕士学位。在加入Life Technologies公司前，他还在三星公司开发了3D图像软件，并在一家创业公司领导软件团队。



在线互动交流平台
官方微博：http://weibo.com / cmpjsj
机工社通信书友QQ群 476980987

获取更多资源及图书信息请关注

地址：北京市百万庄大街22号
邮政编码：100037
电话服务
服务咨询热线：010-88379833
读者购书热线：010-88379649
网络服务
机工官网：www.cmpbook.com
机工官博：weibo.com/cmp1952
教育服务网：www.cmpedu.com
金书网：www.golden-book.com
封面无防伪标均为盗版

策划编辑◎李馨馨



机械工业出版社微信服务号


计算机分社微信服务号


机械社电气信息图书公众号



ISBN 978-7-111-52904-0
定价：59.00元